US009110657B2

(12) **United States Patent**
Yap

(10) **Patent No.: US 9,110,657 B2**
(45) **Date of Patent: Aug. 18, 2015**

(54) **FLOWCHART COMPILER FOR A COMPOUND COMPLEX INSTRUCTION SET COMPUTER (CCISC) PROCESSOR ARCHITECTURE**

(71) Applicant: **Tom Yap**, Centennial, CO (US)

(72) Inventor: **Tom Yap**, Centennial, CO (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 120 days.

(21) Appl. No.: **13/746,249**

(22) Filed: **Jan. 21, 2013**

(65) **Prior Publication Data**

US 2014/0208081 A1 Jul. 24, 2014

(51) **Int. Cl.**
G06F 9/30 (2006.01)

(52) **U.S. Cl.**
CPC ***G06F 9/30145*** (2013.01)

(58) **Field of Classification Search**
USPC 717/140
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,569,016 A	2/1986	Hao et al.	
4,589,065 A	5/1986	Auslander et al.	
4,589,087 A	5/1986	Auslander et al.	
4,947,316 A	8/1990	Fisk et al.	
5,307,492 A	4/1994	Benson	
5,371,860 A	12/1994	Mura et al.	
5,471,593 A	11/1995	Branigin	
5,544,342 A	8/1996	Dean	
5,598,546 A *	1/1997	Blomgren	712/209
6,076,158 A	6/2000	Sites et al.	
6,167,509 A	12/2000	Sites et al.	
6,212,629 B1	4/2001	McFarland et al.	
6,763,452 B1	7/2004	Hohensee et al.	
6,941,545 B1	9/2005	Reese et al.	
7,047,394 B1	5/2006	Dyke et al.	
7,401,333 B2	7/2008	Vandeweerd	
7,404,067 B2	7/2008	Aamodt et al.	
7,409,670 B1	8/2008	Pritchard et al.	
7,421,566 B2	9/2008	Gschwind et al.	
7,478,224 B2	1/2009	Strom et al.	
7,617,496 B2	11/2009	Gonion	
7,818,547 B2	10/2010	Aamodt et al.	
7,873,953 B1	1/2011	Pritchard et al.	
7,941,790 B2	5/2011	Cabillic et al.	
8,006,071 B2	8/2011	Ball	
8,065,502 B2	11/2011	Gonion	
8,065,504 B2	11/2011	Yates et al.	
8,127,121 B2	2/2012	Yates et al.	
8,161,444 B1	4/2012	Pritchard	
8,166,281 B2	4/2012	Gschwind et al.	
8,281,109 B2	10/2012	Valentine et al.	

OTHER PUBLICATIONS

Fryer et al., Microsoft Press Computer Dictionary, Microsoft Press, third ed., 4 pages.*

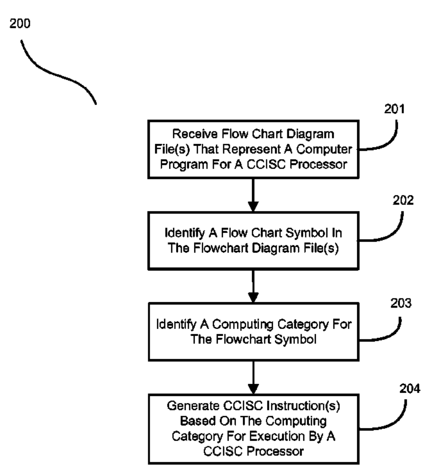
* cited by examiner

Primary Examiner — John Chavis
(74) *Attorney, Agent, or Firm* — Duft Bornsen and Fettig, LLP

(57) **ABSTRACT**

Systems and methods herein provide for a compiler to create executable programs for a compound instruction based processor directly from flowcharts. In one embodiment, a system receives one or more flowchart diagram files that represent a computer program for a Compound CISC (CCISC) processor. The system identifies a flowchart symbol in the one or more flowchart diagram files, identifies a computing category for the flowchart symbol, and generates one or more CCISC instructions based on the computing category for execution by the CCISC processor. Further, the one or more CCISC instructions generated by the flowchart compiler direct the CCISC processor to access and operate on at least two data values in a multi-channel memory during the same clock cycle.

18 Claims, 37 Drawing Sheets



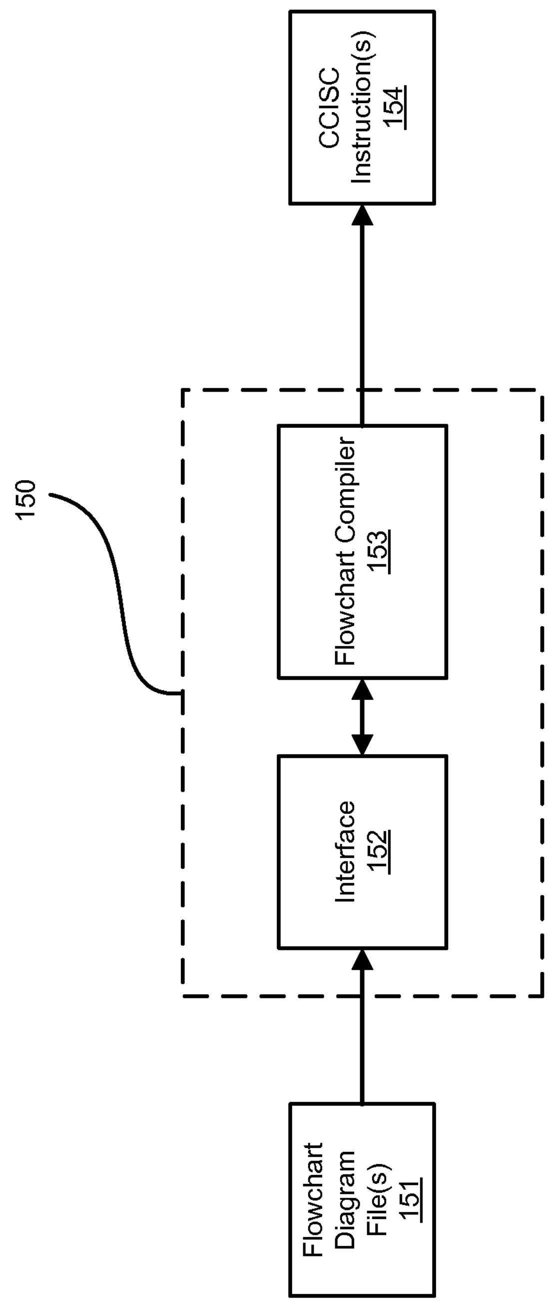

150
Flowchart Diagram File(s) 151
Interface 152
Flowchart Compiler 153
CCISC Instruction(s) 154

FIG. 1

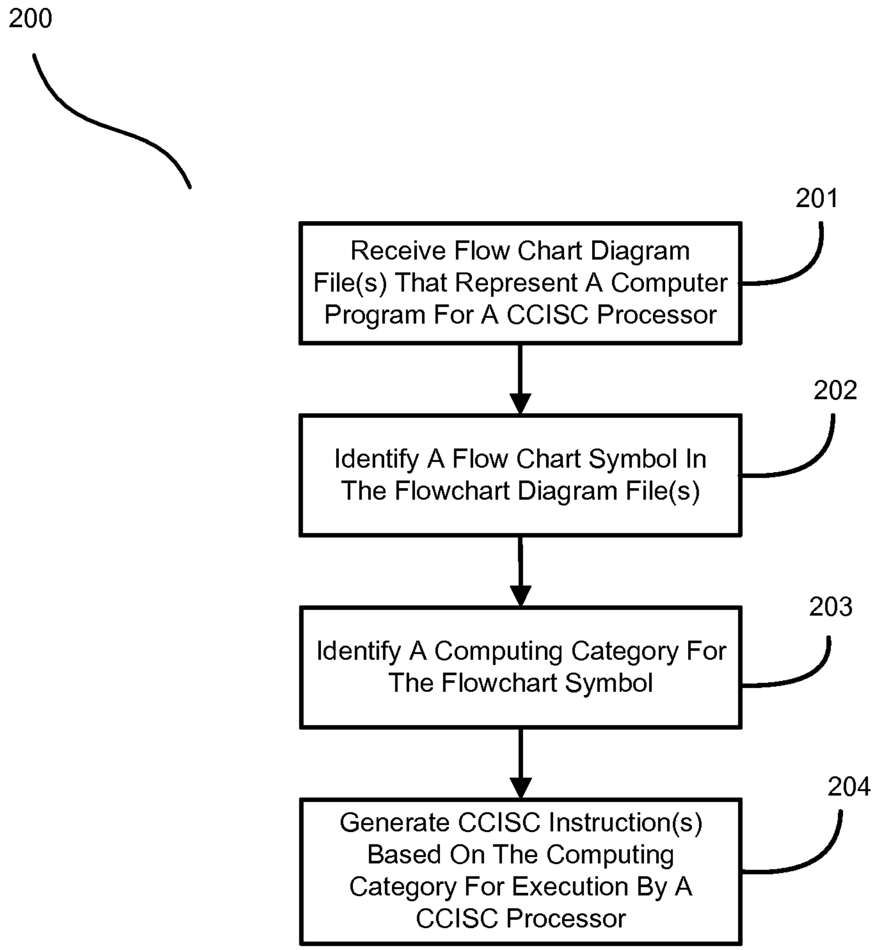

200
201
Receive Flow Chart Diagram File(s) That Represent A Computer Program For A CCISC Processor
202
Identify A Flow Chart Symbol In The Flowchart Diagram File(s)
203
Identify A Computing Category For The Flowchart Symbol
204
Generate CCISC Instruction(s) Based On The Computing Category For Execution By A CCISC Processor

FIG. 2

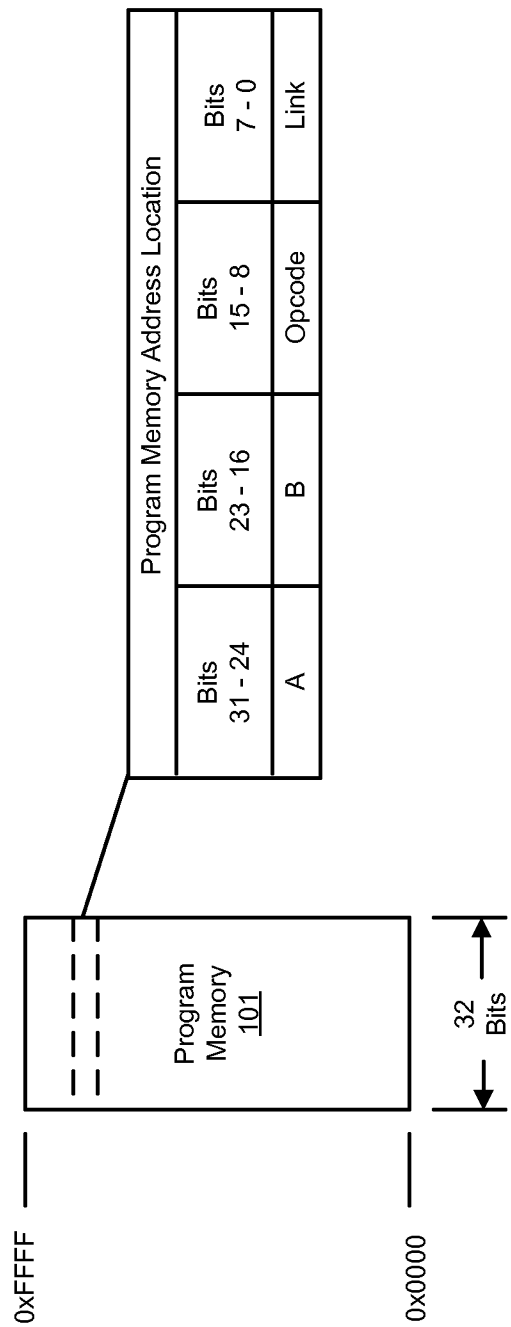

Program Memory Address Location
Bits 31 - 24
A
Bits 23 - 16
B
Bits 15 - 8
Opcode
Bits 7 - 0
Link
Program Memory 101
0xFFFF
0x0000
32 Bits

FIG. 3

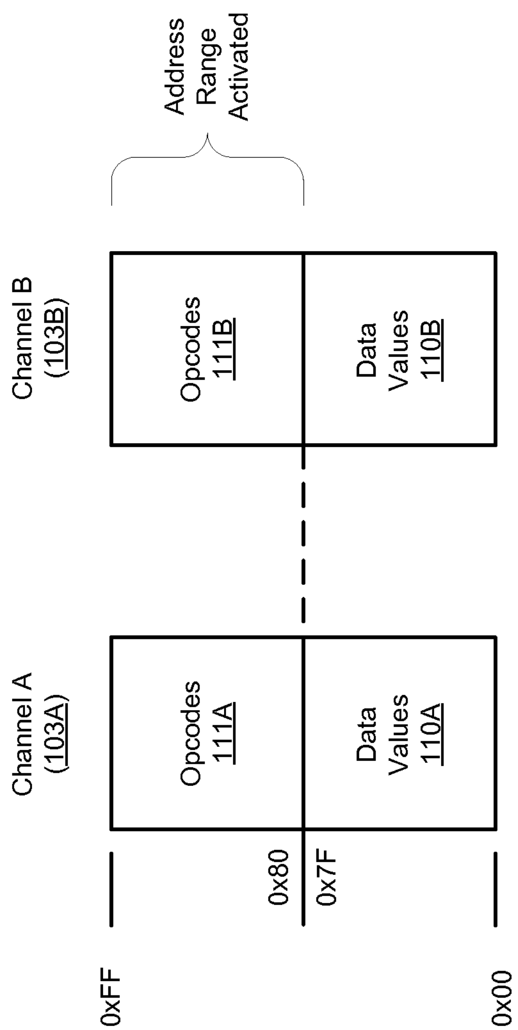

Address Range Activated
Channel B (103B)
Opcodes 111B
Data Values 110B
Channel A (103A)
Opcodes 111A
Data Values 110A
0xFF
0x80
0x7F
0x00

FIG. 4

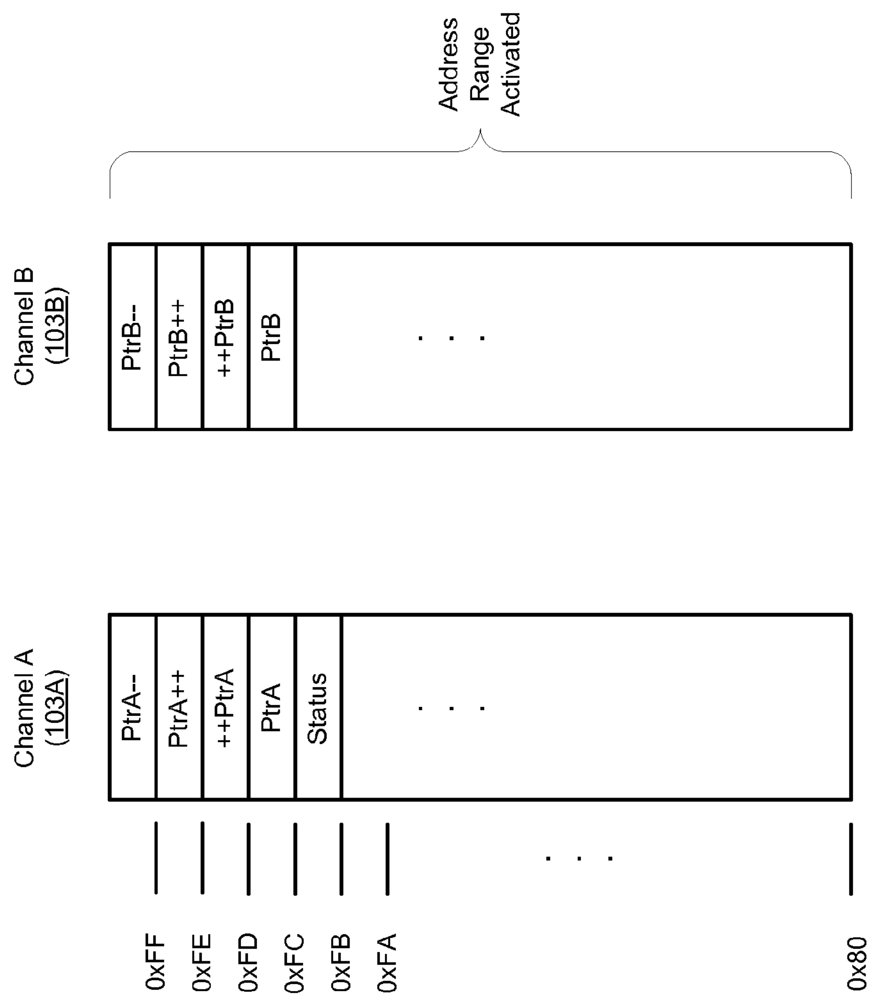

Address Range Activated
Channel B (103B)
PtrB--
PtrB++
++PtrB
PtrB
Channel A (103A)
PtrA--
PtrA++
++PtrA
PtrA
Status
0xFF
0xFE
0xFD
0xFC
0xFB
0xFA
0x80

FIG. 5

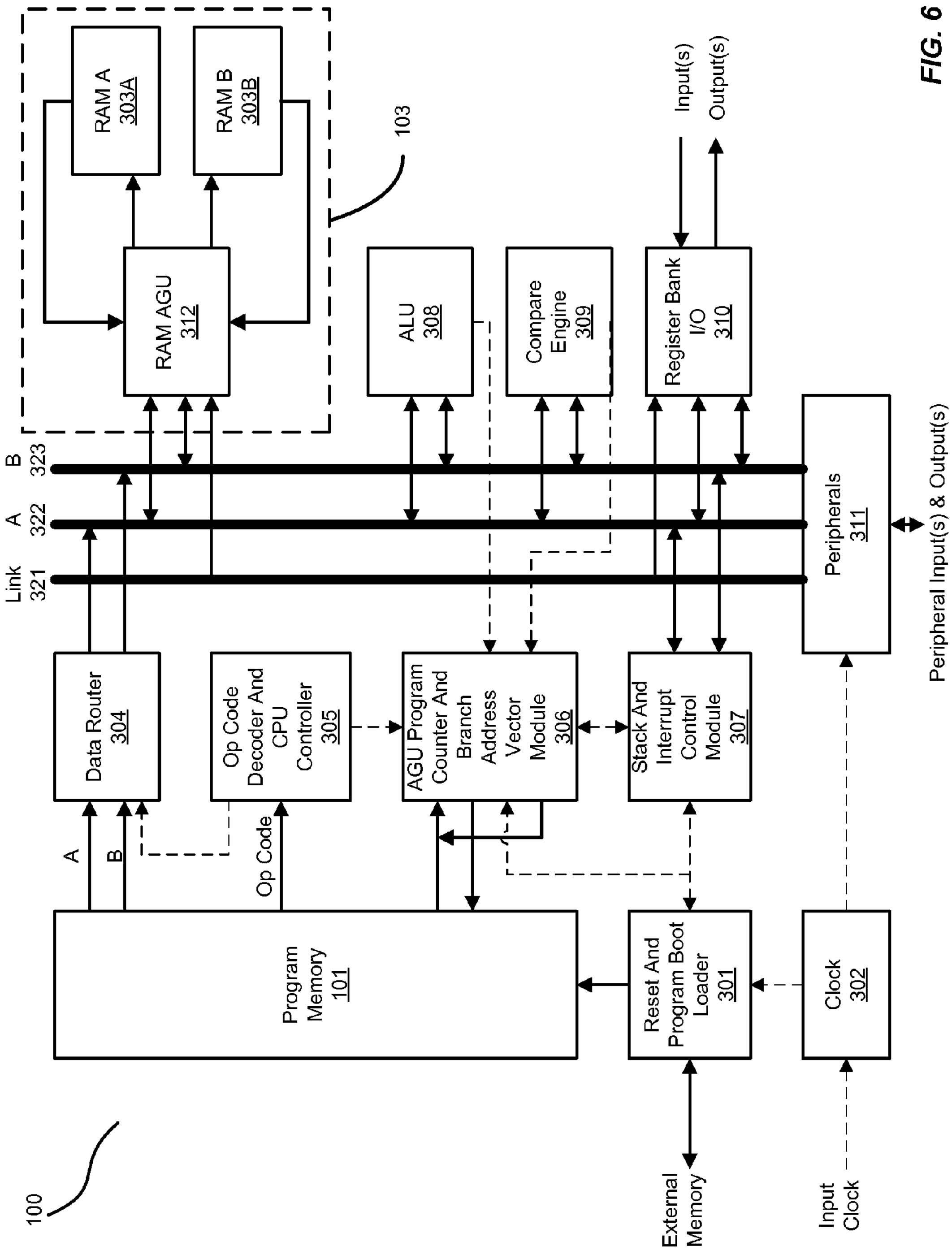

RAM A
303A
RAM B
303B
103
RAM AGU
312
ALU
308
Compare
Engine
309
Register Bank
I/O
310
Input(s)
Output(s)
B
323
A
322
Link
321
Peripherals
311
Peripheral Input(s) & Output(s)
Data Router
304
Op Code
Decoder And
CPU
Controller
305
AGU Program
Counter And
Branch
Address
Vector
Module
306
Stack And
Interrupt
Control
Module
307
A
B
Op Code
Program
Memory
101
Reset And
Program Boot
Loader
301
Clock
302
External
Memory
Input
Clock
100

FIG. 6

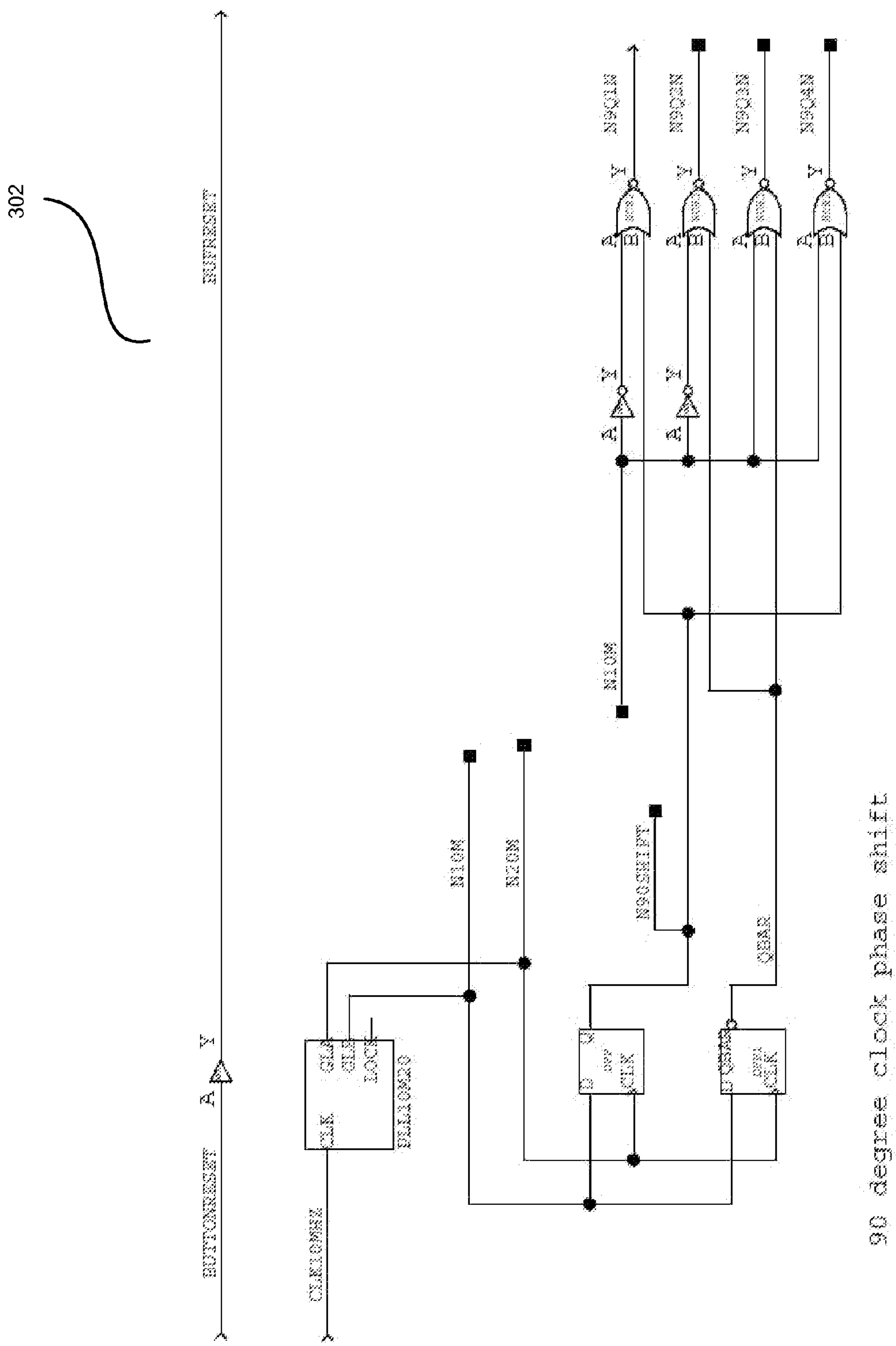

302
BUTTONRESET
CLK10MHZ
N10M
N20M
N90SHIFT
QBAR
N10M
N9Q1N
N9Q2N
N9Q3N
N9Q4N
90 degree clock phase shift

FIG. 7

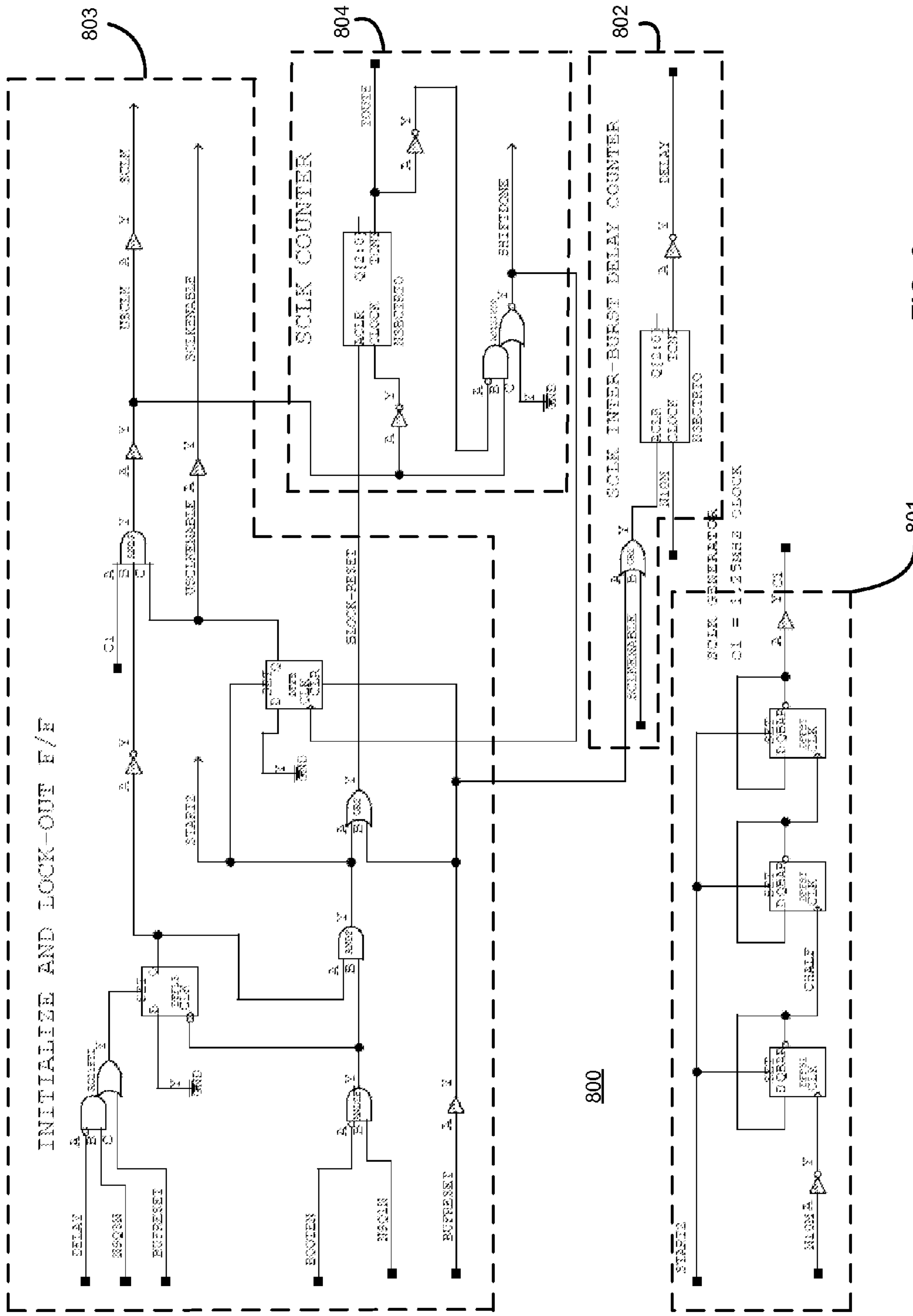

803
804
802
801
800
INITIALIZE AND LOCK-OUT F/F
SCLK COUNTER
SCLK INTER-BURST DELAY COUNTER
SCLK GENERATOR
C1 = 1.25MHz CLOCK
SCLKENABLE
SHIFTDONE
DELAY
START2
BUFRESET
C1

FIG. 8

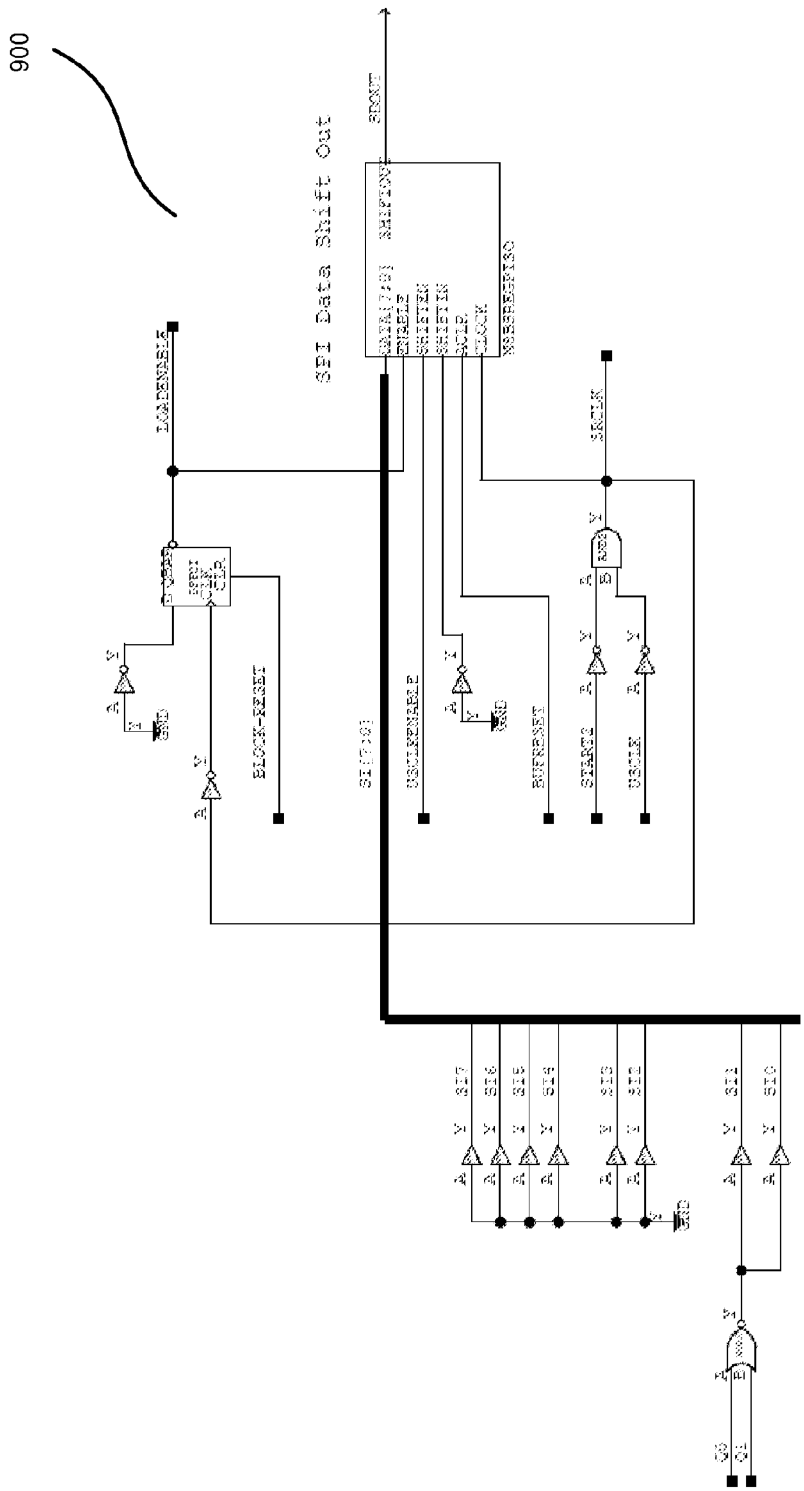


FIG. 9

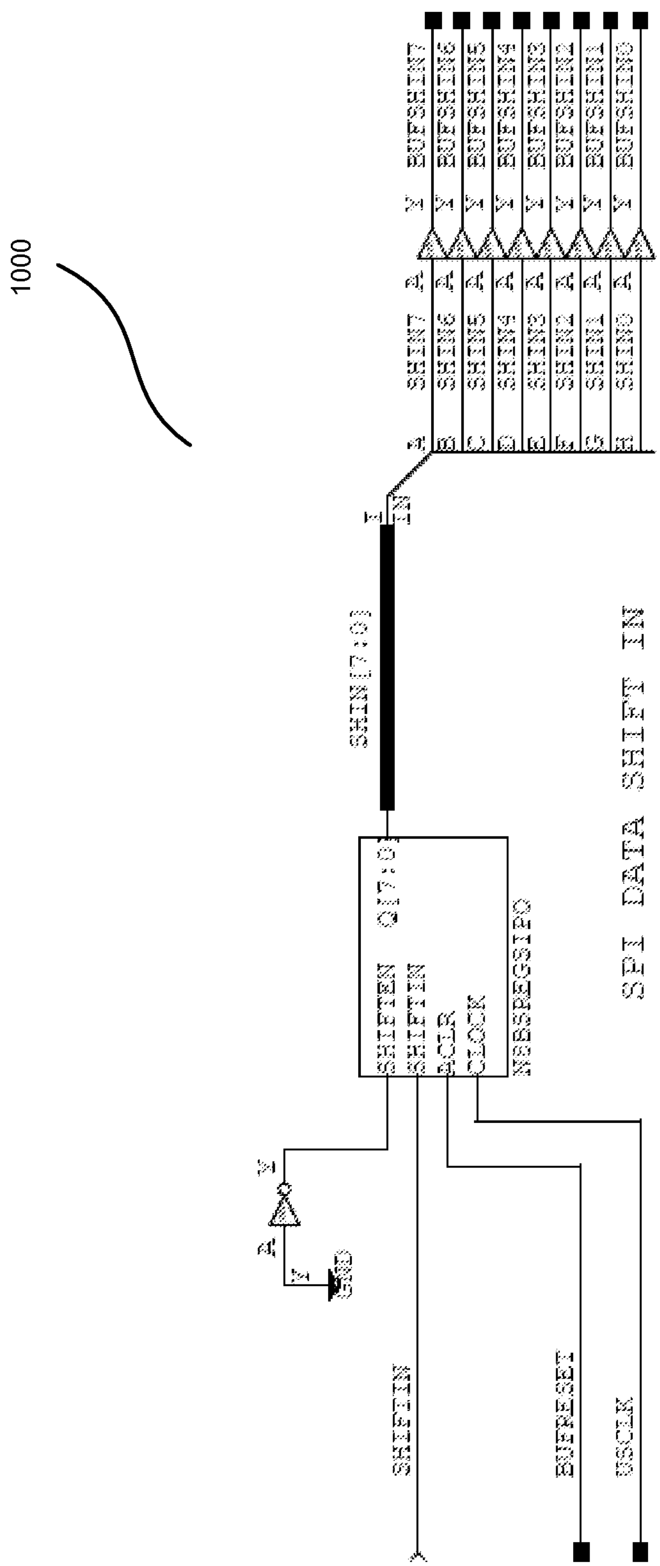


FIG. 10

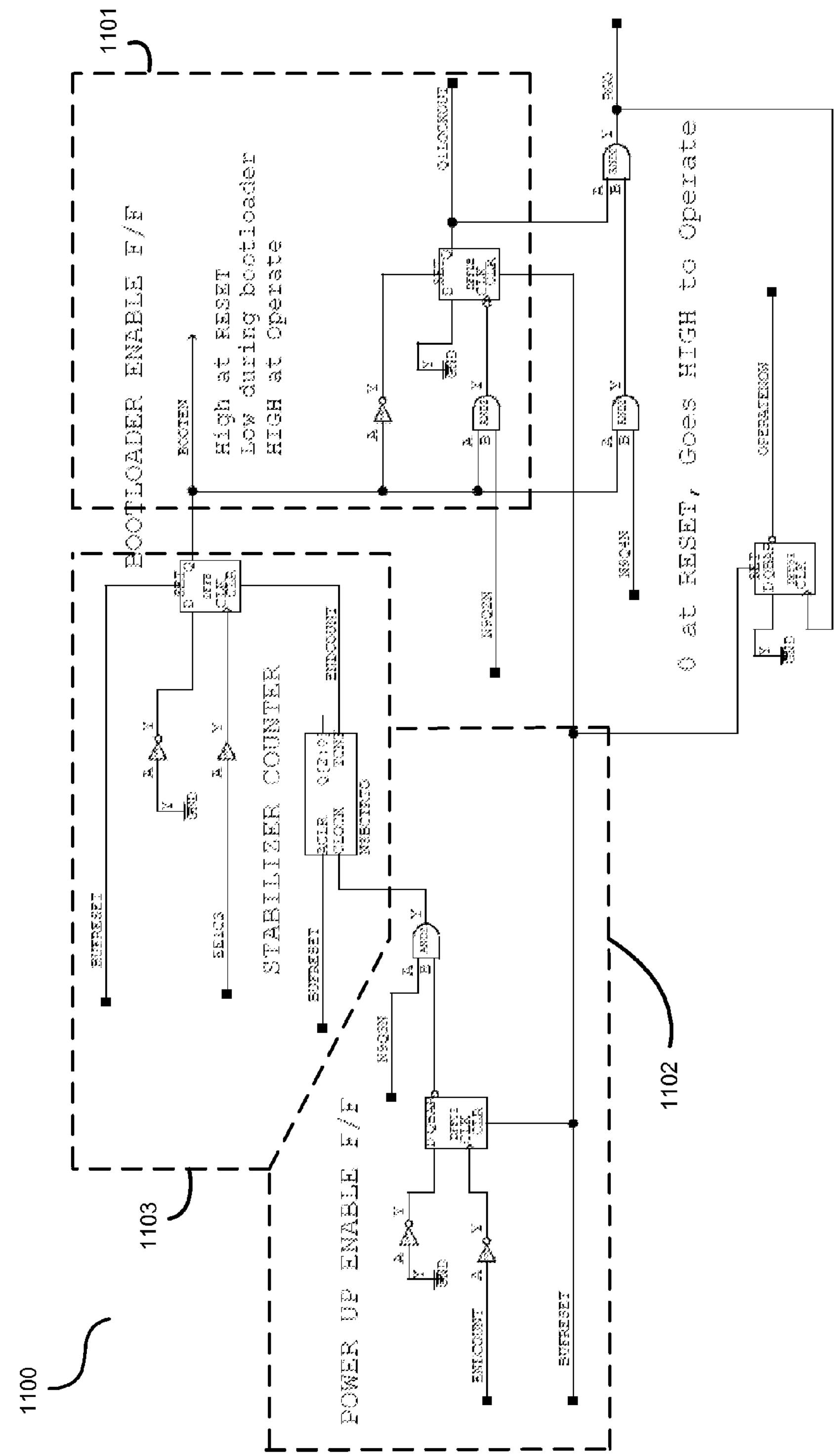

1100
1101
1102
1103
BOOTLOADER ENABLE F/F
High at RESET
Low during bootloader
HIGH at Operate
STABILIZER COUNTER
POWER UP ENABLE F/F
0 at RESET, Goes HIGH to Operate

FIG. 11

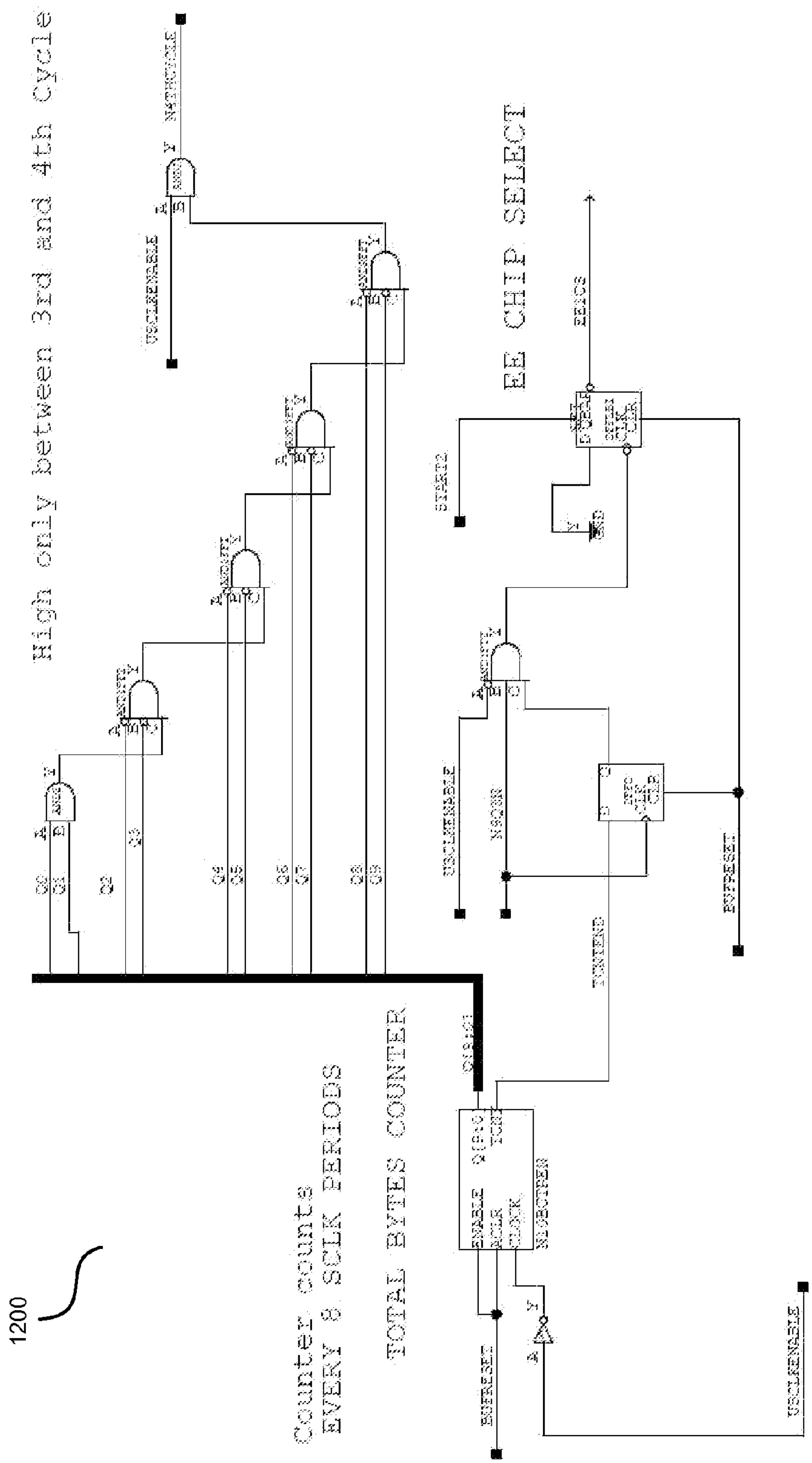

High only between 3rd and 4th Cycle
EE CHIP SELECT
Counter counts
EVERY 8 SCLK PERIODS
TOTAL BYTES COUNTER
1200

FIG. 12

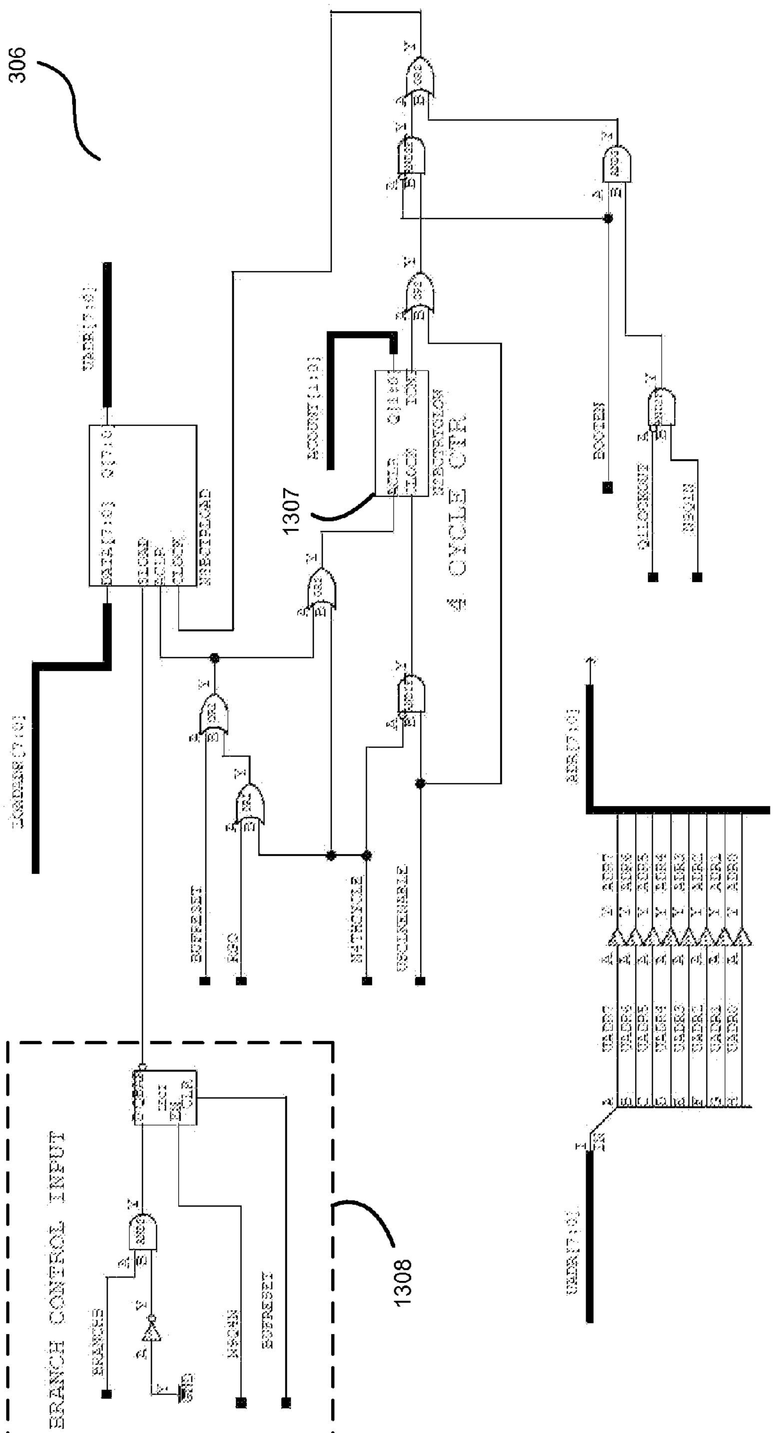


FIG. 13

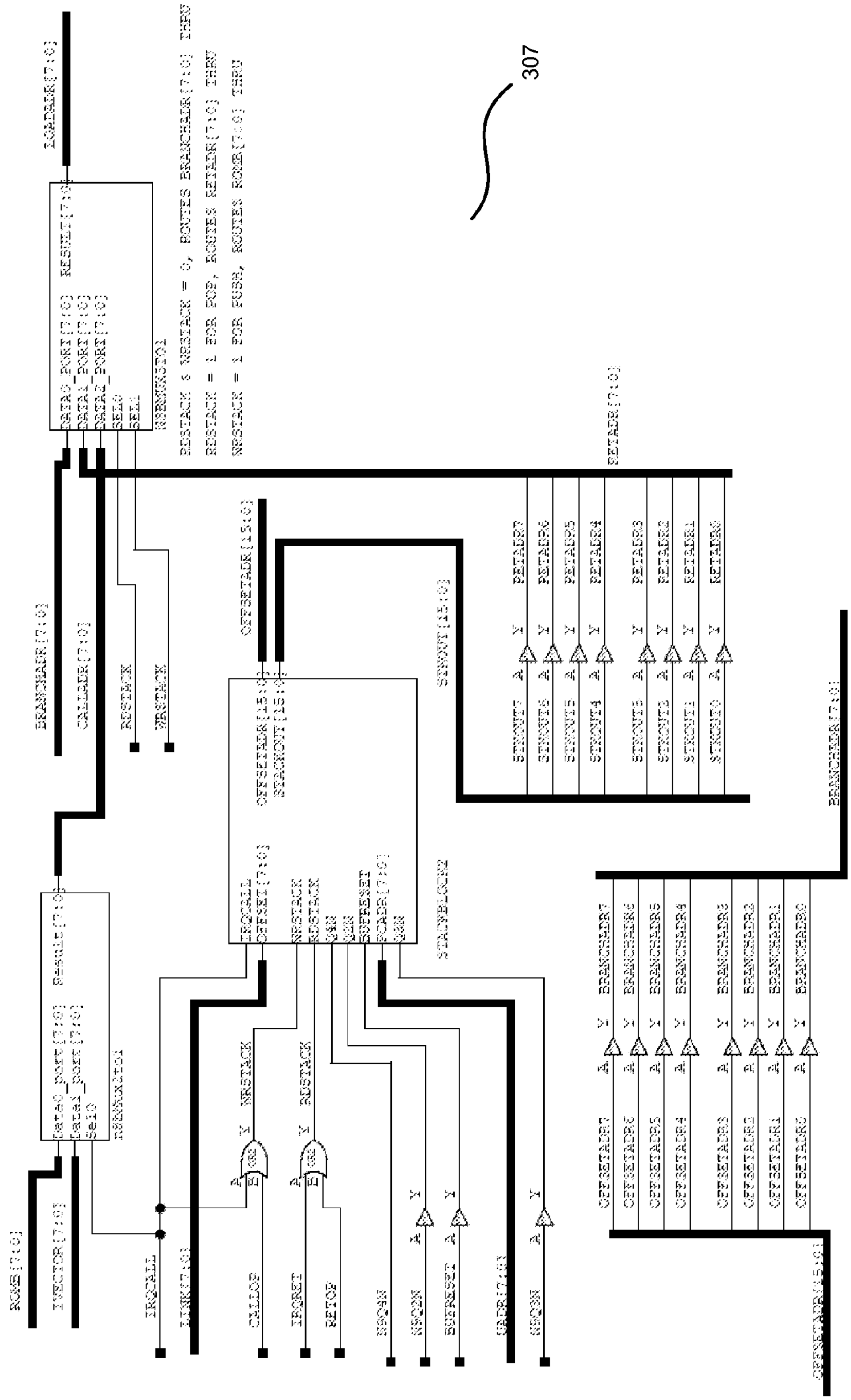

307
LOADADR[7:0]
RDSTACK & WRSTACK = 0, ROUTES BRANCHADR[7:0] THRU
RDSTACK = 1 FOR POP, ROUTES RETADR[7:0] THRU
WRSTACK = 1 FOR PUSH, ROUTES ROMB[7:0] THRU
BRANCHADR[7:0]
CALLADR[7:0]
RDSTACK
WRSTACK
OFFSETADR[15:0]
STKOUT[15:0]
RETADR[7:0]
STACKBLOCK2
ROMB[7:0]
IVECTOR[7:0]
IRQCALL
LINK[7:0]
CALLOP
IRQRET
RETOP
SRQ2N
SRESET
PCADR[7:0]

FIG. 14

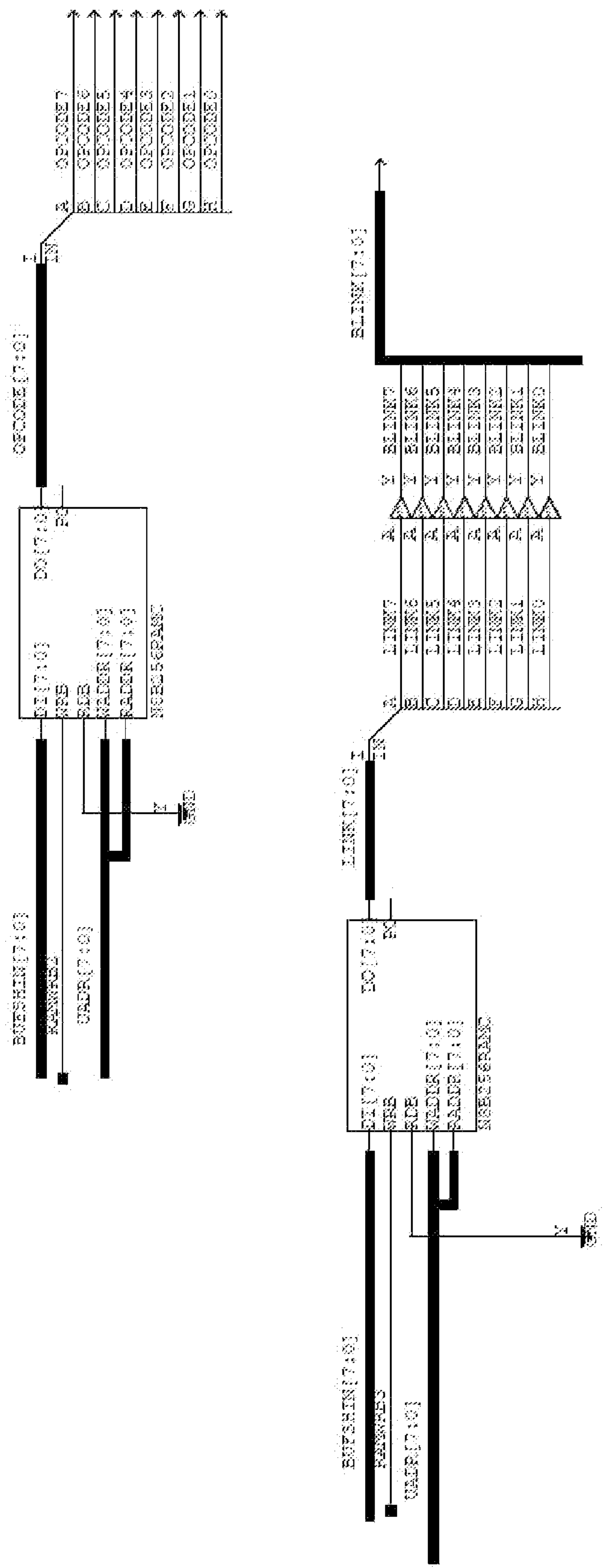


FIG. 15

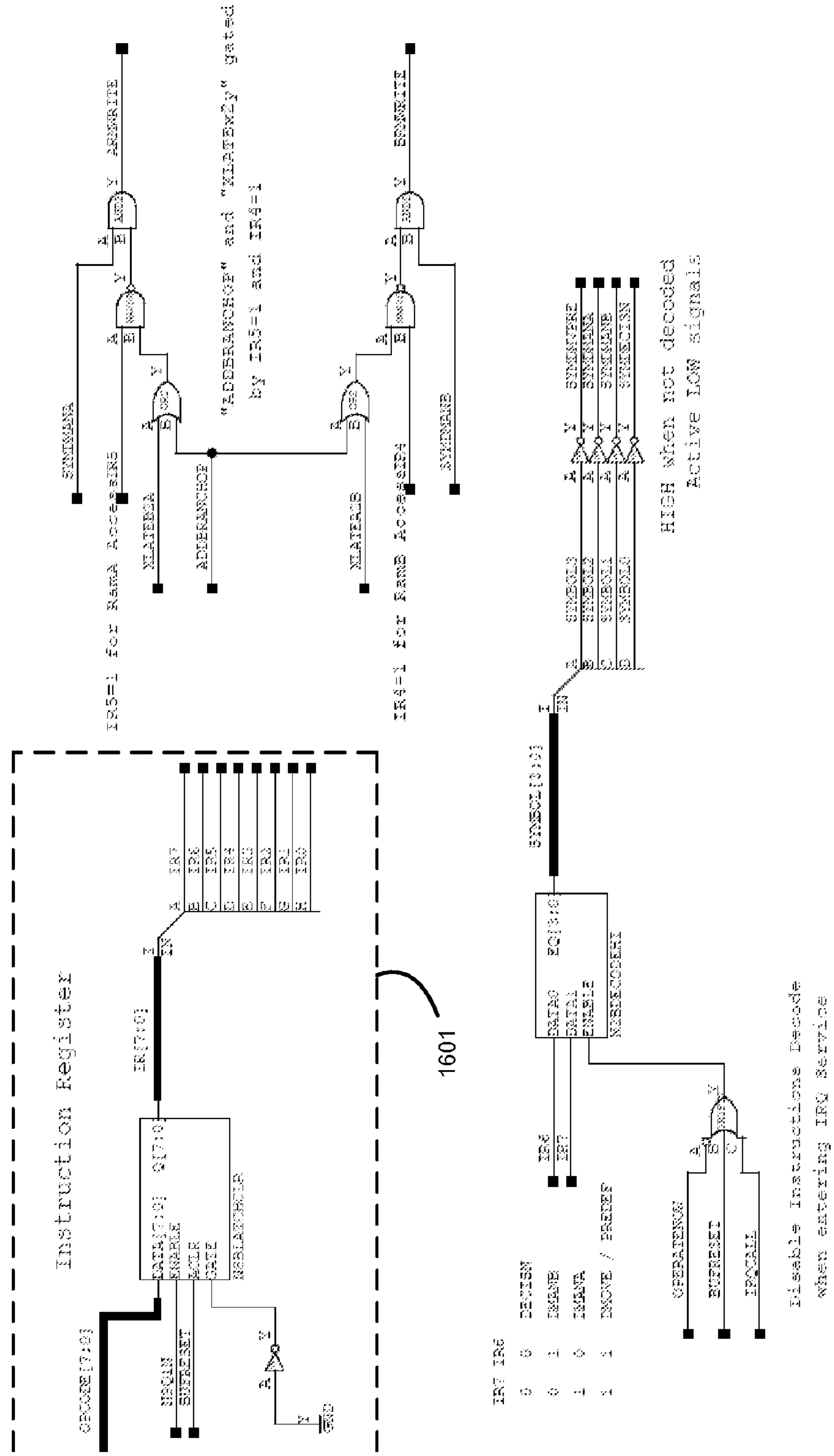
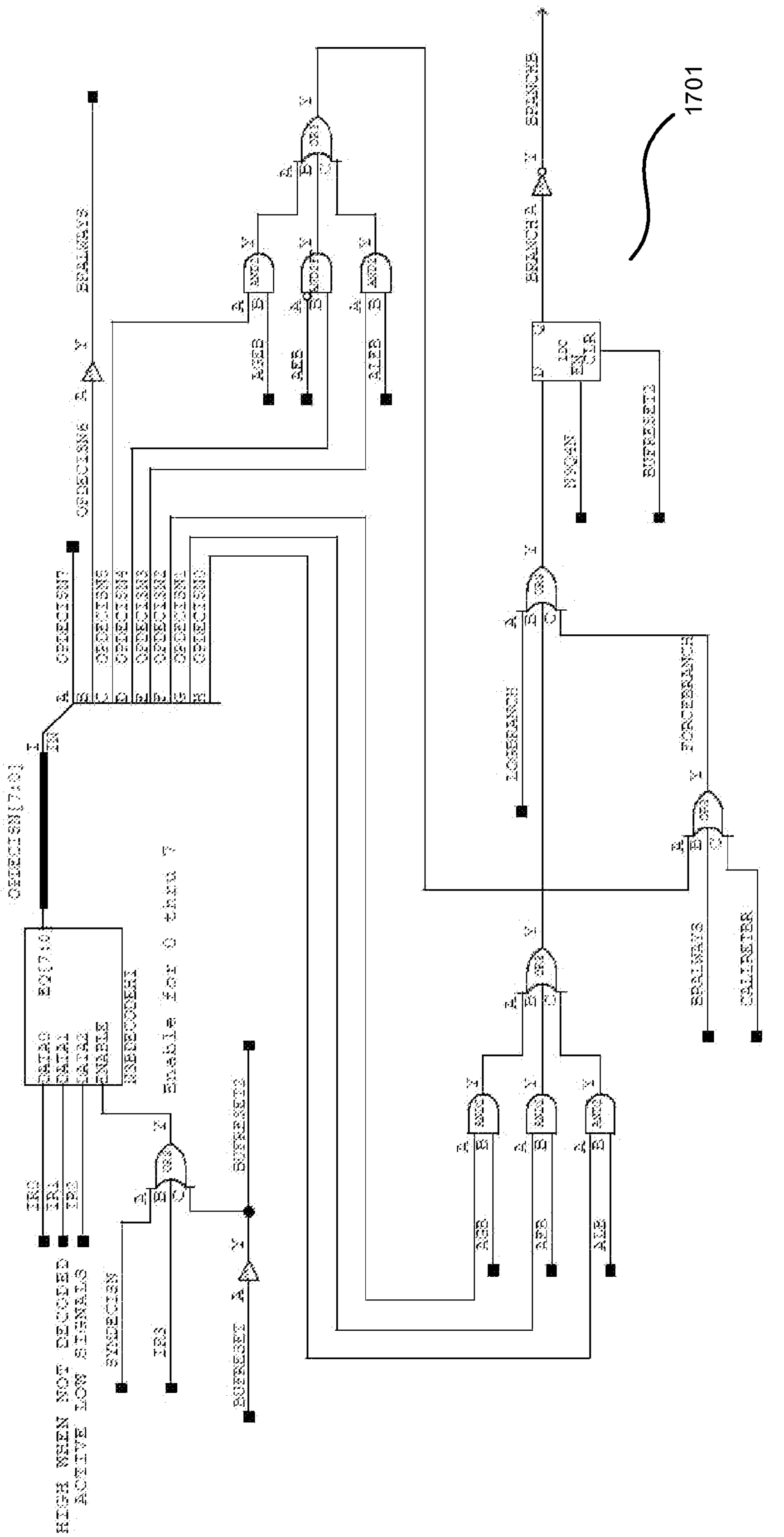

Instruction Register
1601
HIGH when not decoded
Active LOW signals
Disable Instructions Decode
when entering IRQ Service

FIG. 16



1701
HIGH WHEN NOT DECODED
ACTIVE LOW SIGNALS
Enable for 0 thru 7

FIG. 17

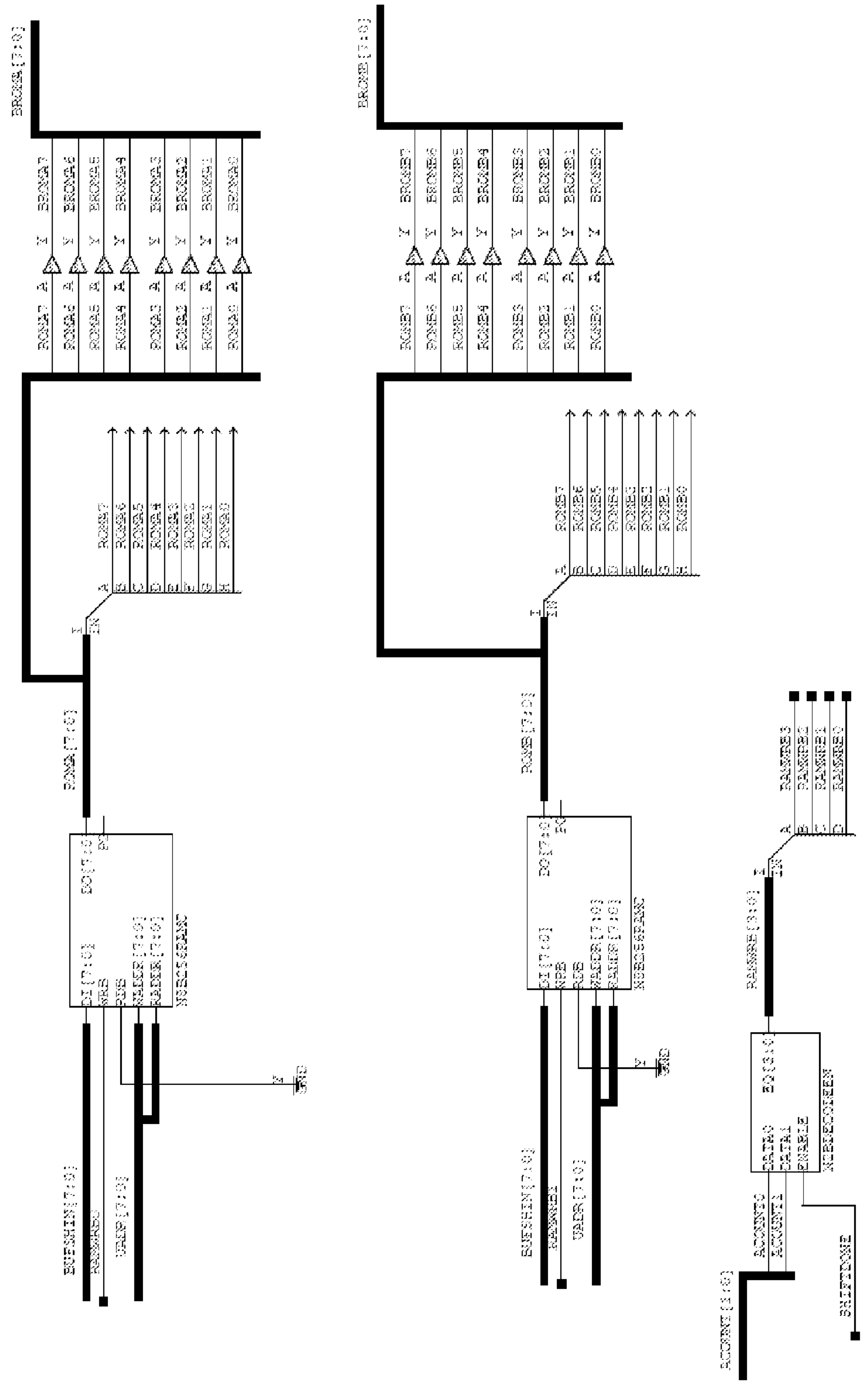


FIG. 18

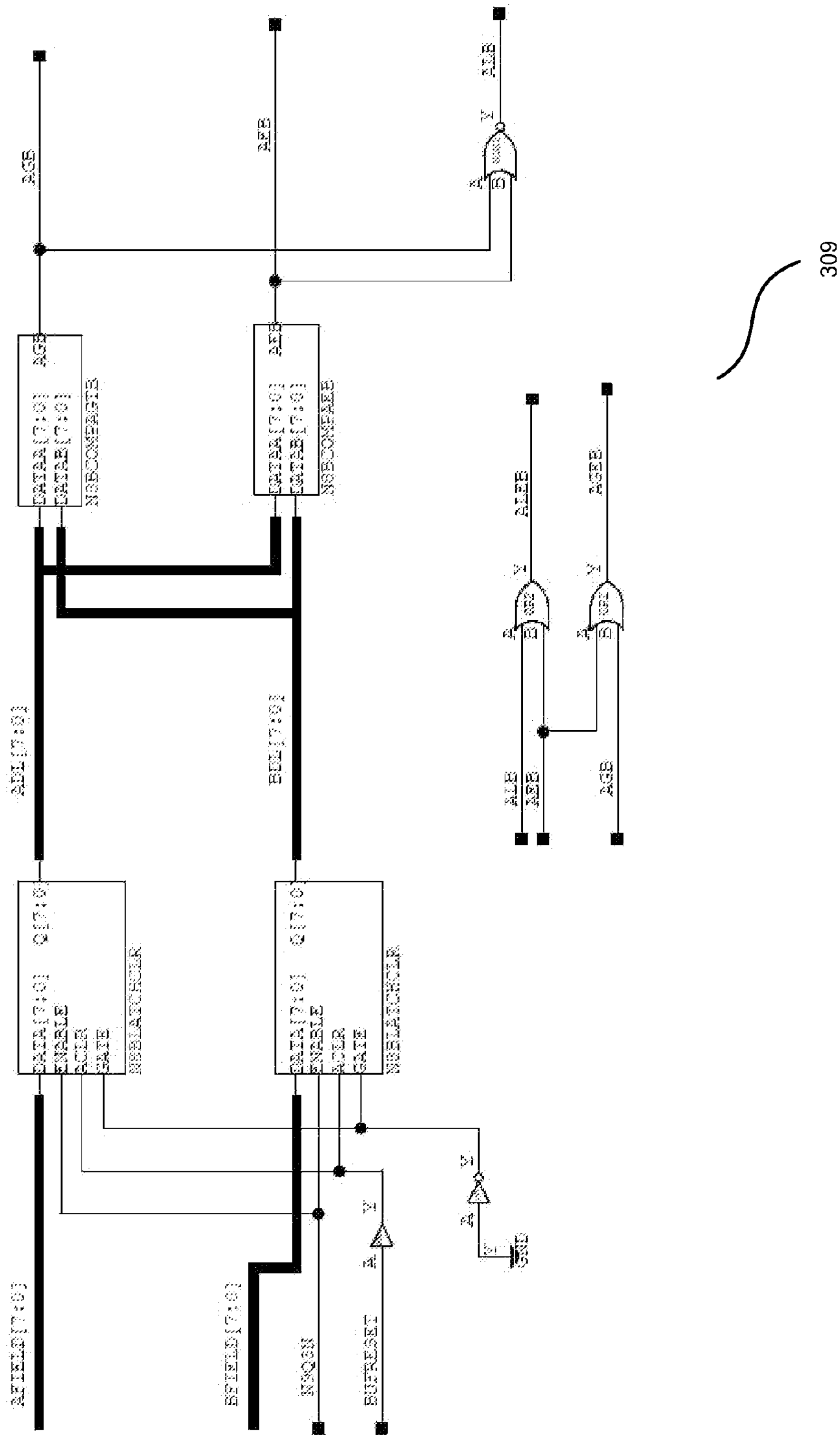


FIG. 19

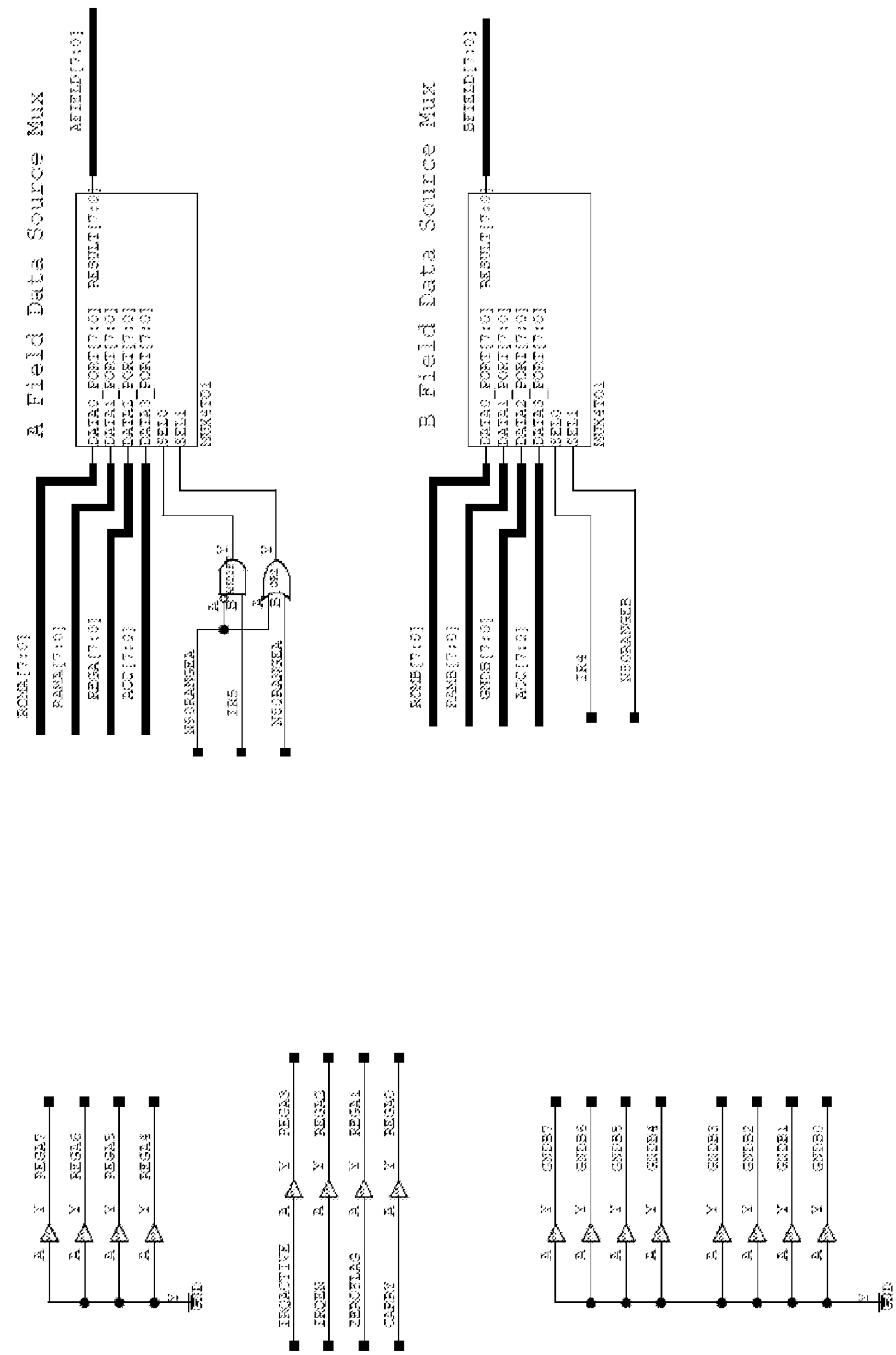

A Field Data Source Mux
AFIELD[7:0]
RESULT[7:0]
ROMA[7:0]
RAMA[7:0]
REGA[7:0]
ACC[7:0]
IR5
IR4
B Field Data Source Mux
BFIELD[7:0]
ROMB[7:0]
RAMB[7:0]
GNDB[7:0]
ACC[7:0]

FIG. 20

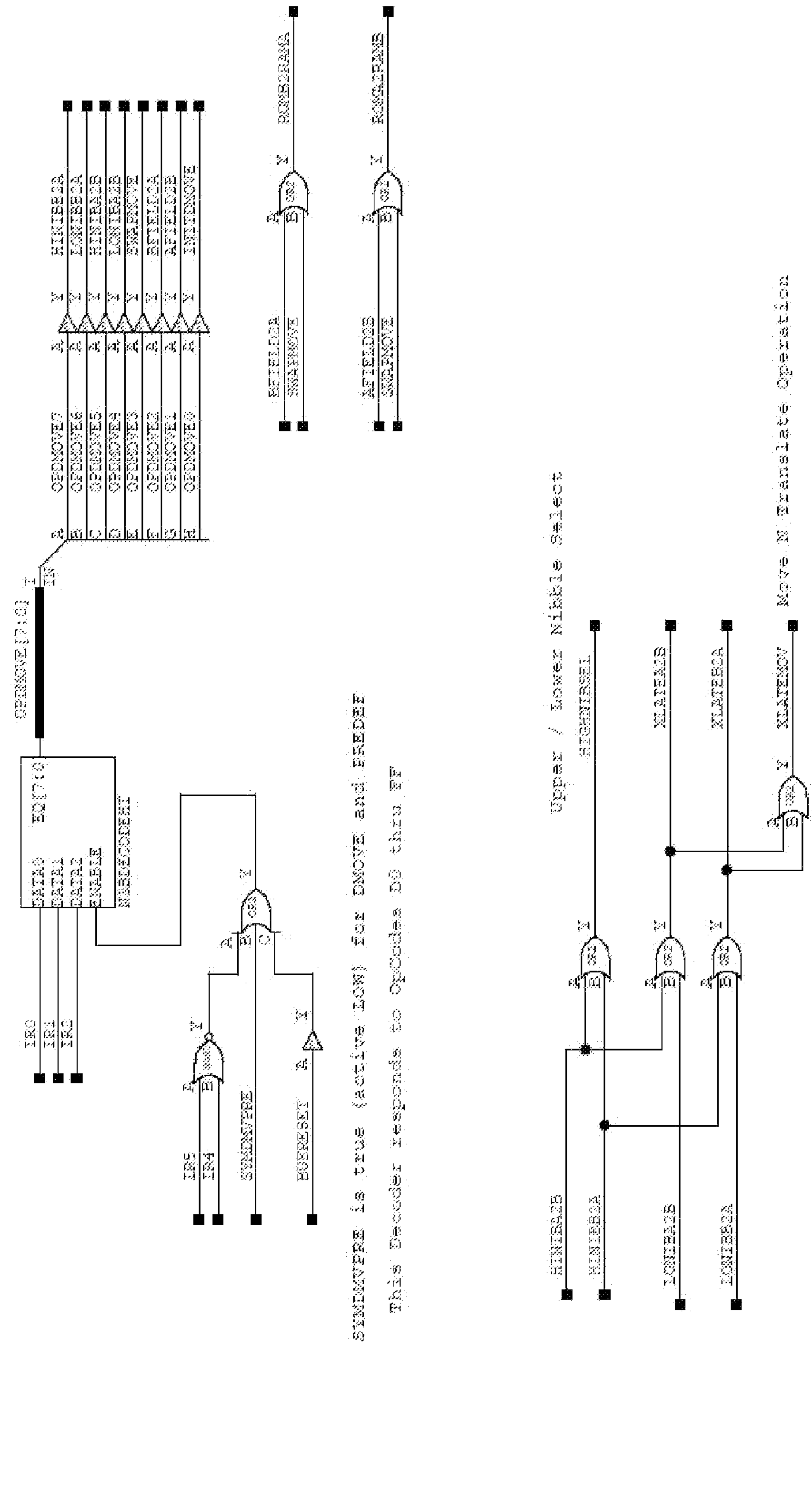


FIG. 21

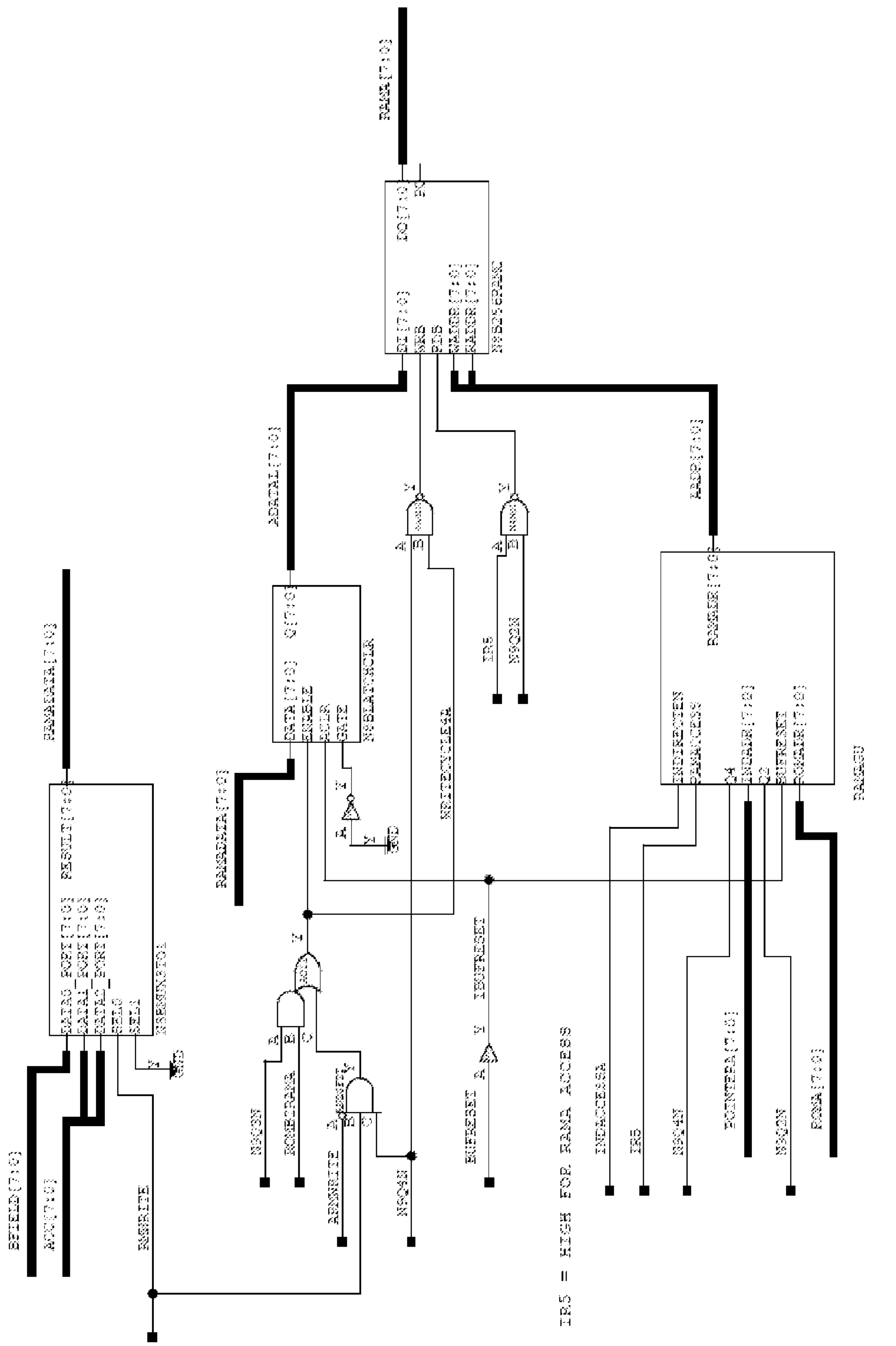


FIG. 22

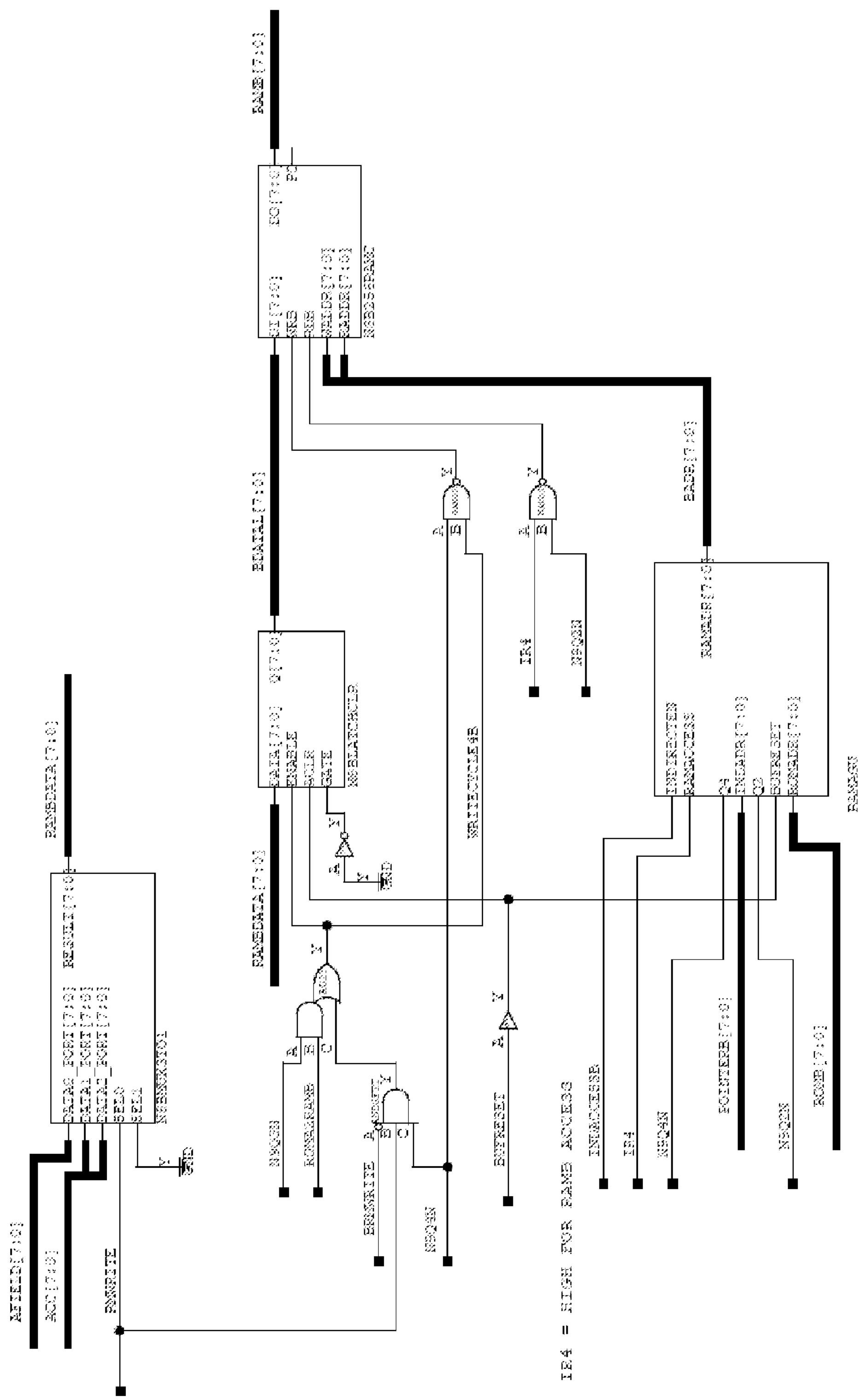


FIG. 23

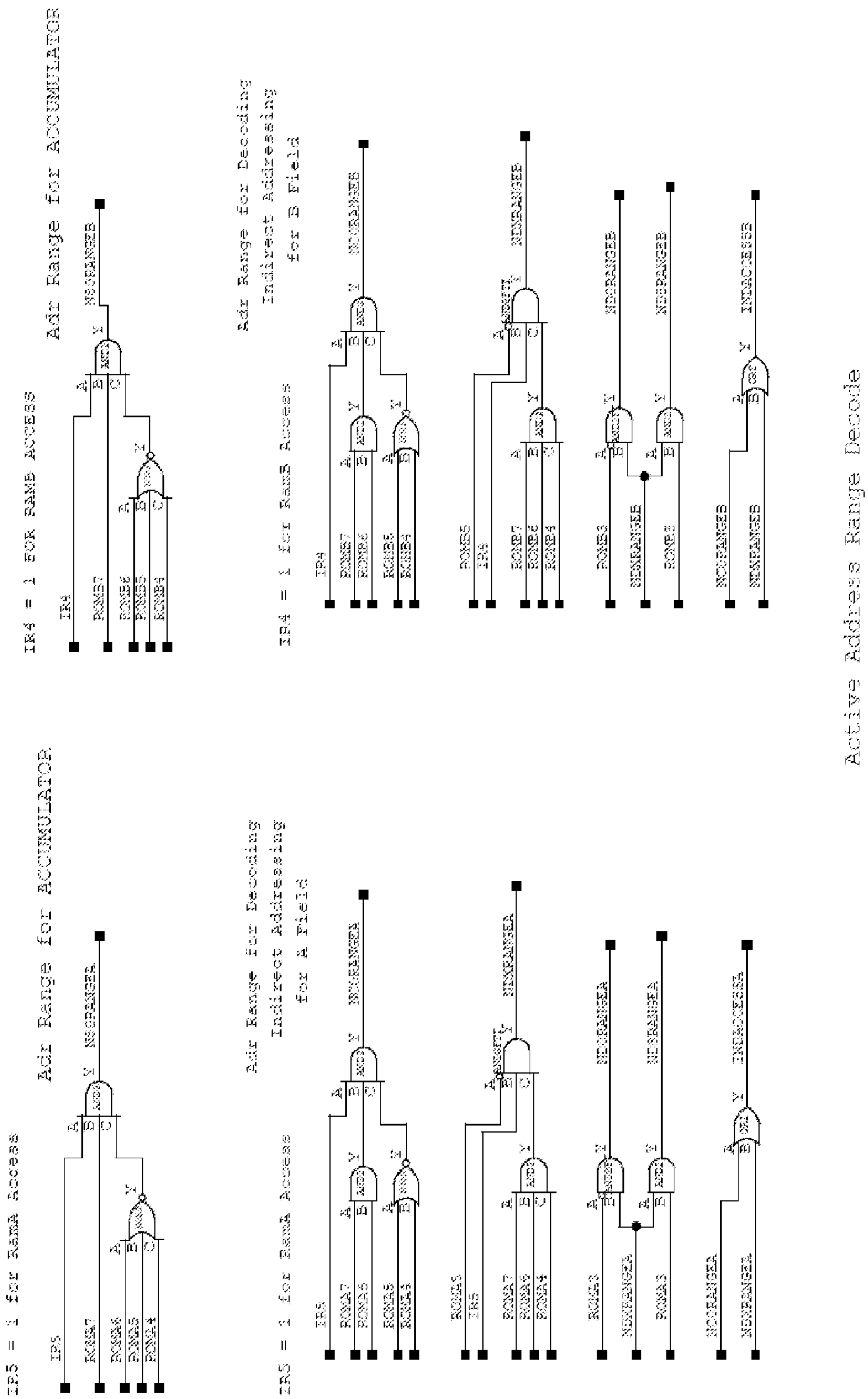

Adr Range for ACCUMULATOR
IR4 = 1 FOR RAMB ACCESS
Adr Range for Decoding Indirect Addressing for B Field
IR4 = 1 for RamB Access
Adr Range for ACCUMULATOR
IR5 = 1 for RamA Access
Adr Range for Decoding Indirect Addressing for A Field
IR5 = 1 for RamA Access
Active Address Range Decode

FIG. 24

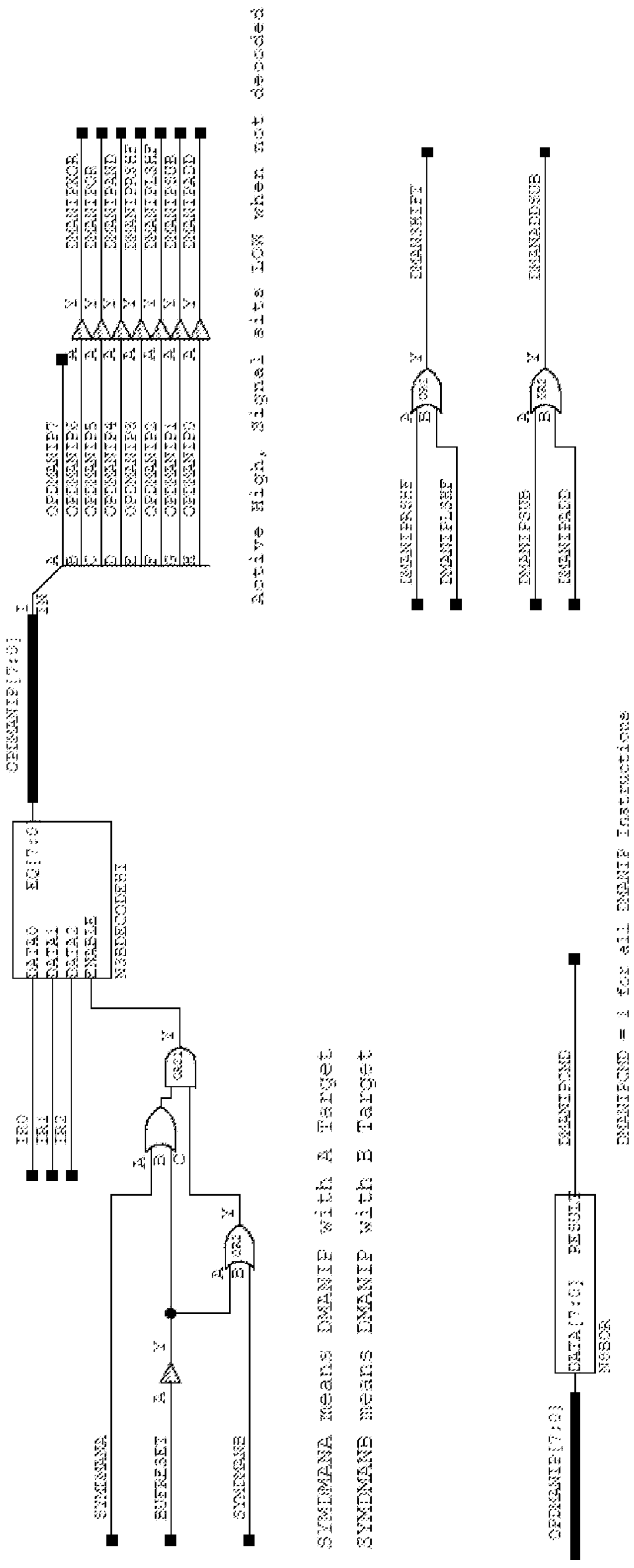

Active High, signal bits LOW when not decoded
SYMDMANA means DMANIP with A Target
SYMDMANB means DMANIP with B Target
DMANIPCMD = 1 for all DMANIP Instructions

FIG. 25

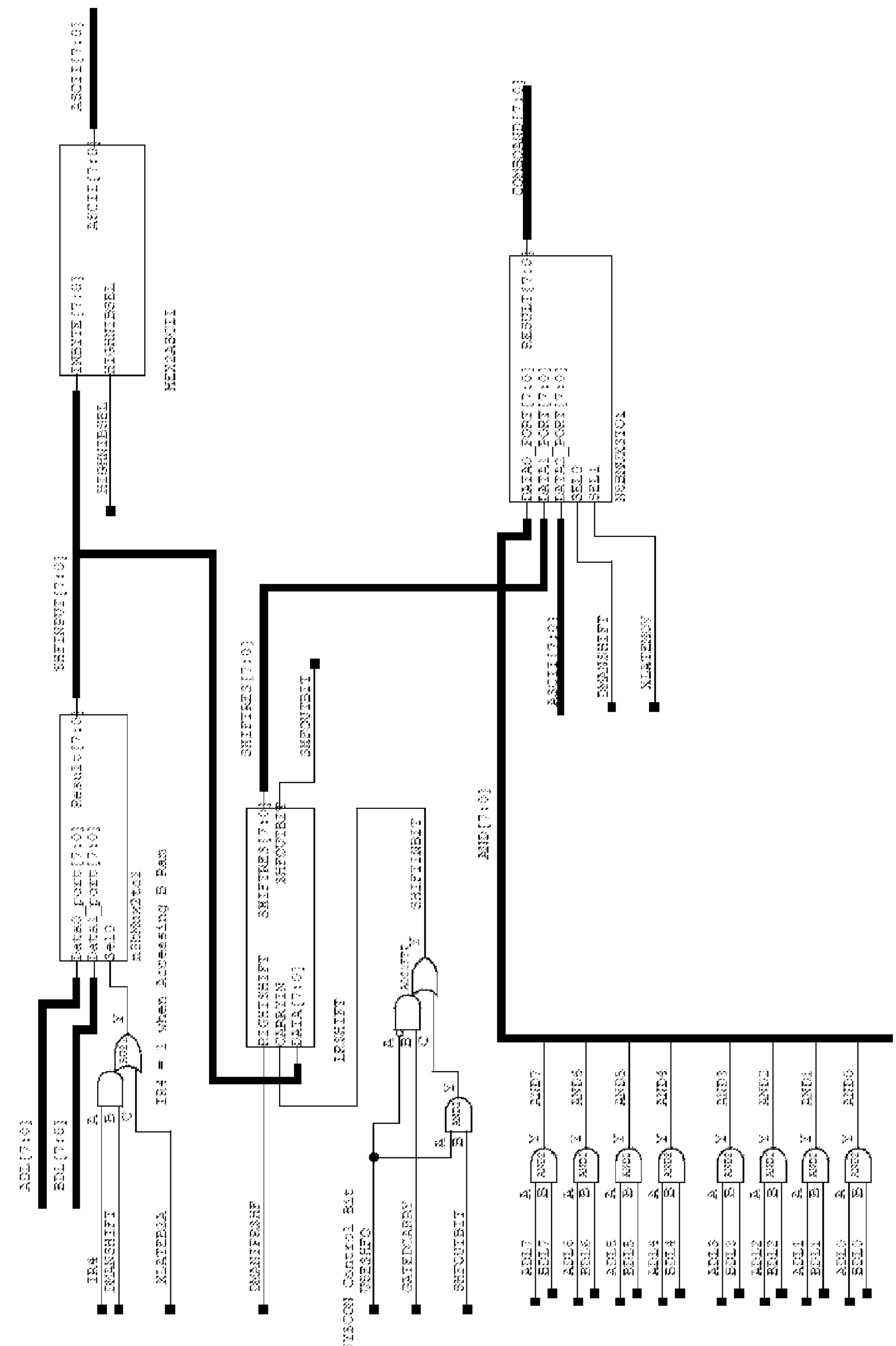


FIG. 26

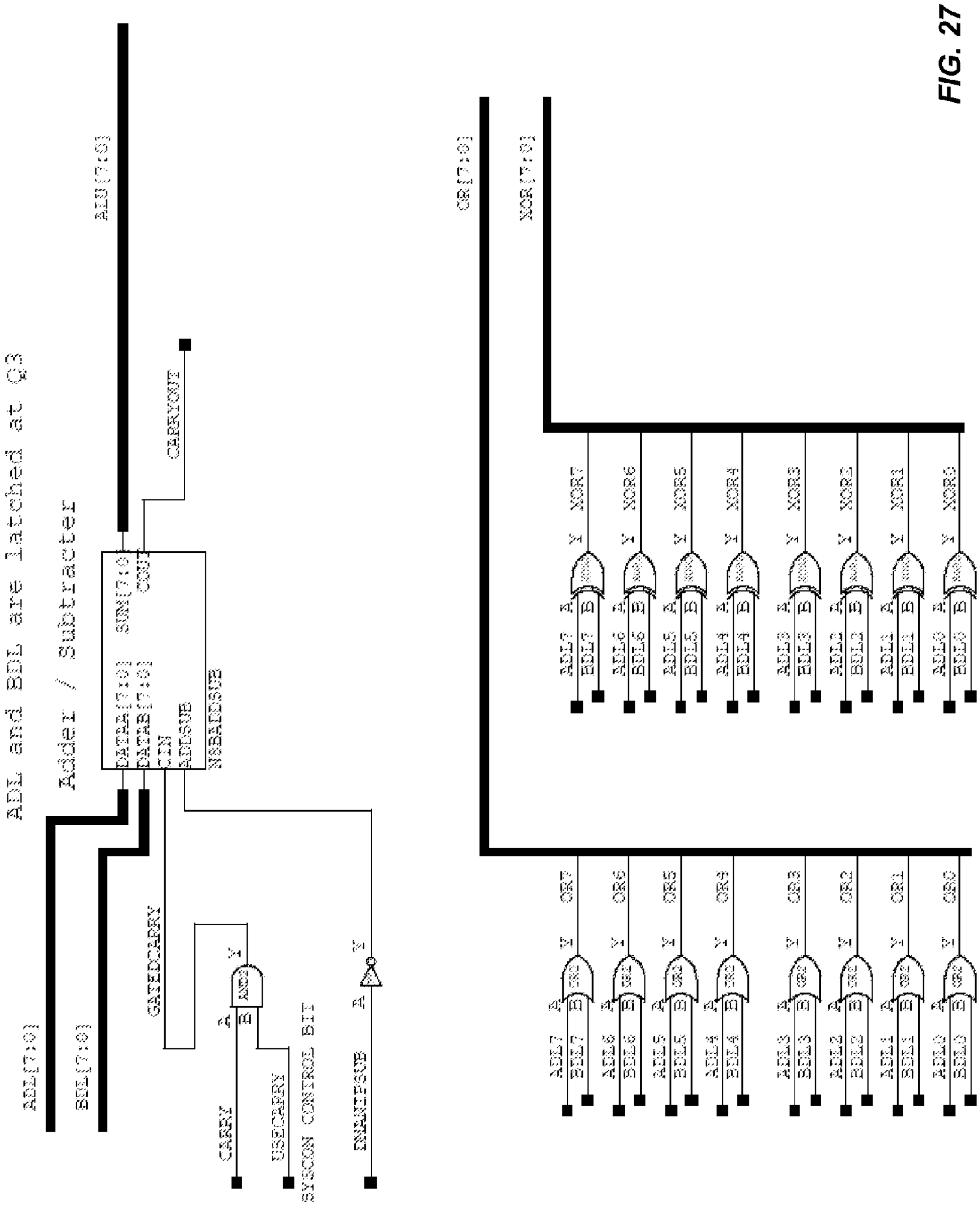

ADL and BDL are latched at Q3
Adder / Subtracter
ALU[7:0]
CARRYOUT
OR[7:0]
XOR[7:0]
ADL[7:0]
BDL[7:0]
GATEDCARRY
CARRY
USECARRY
SYSCON CONTROL BIT
SUM[7:0]
COUT
CIN
ADDSUB

FIG. 27

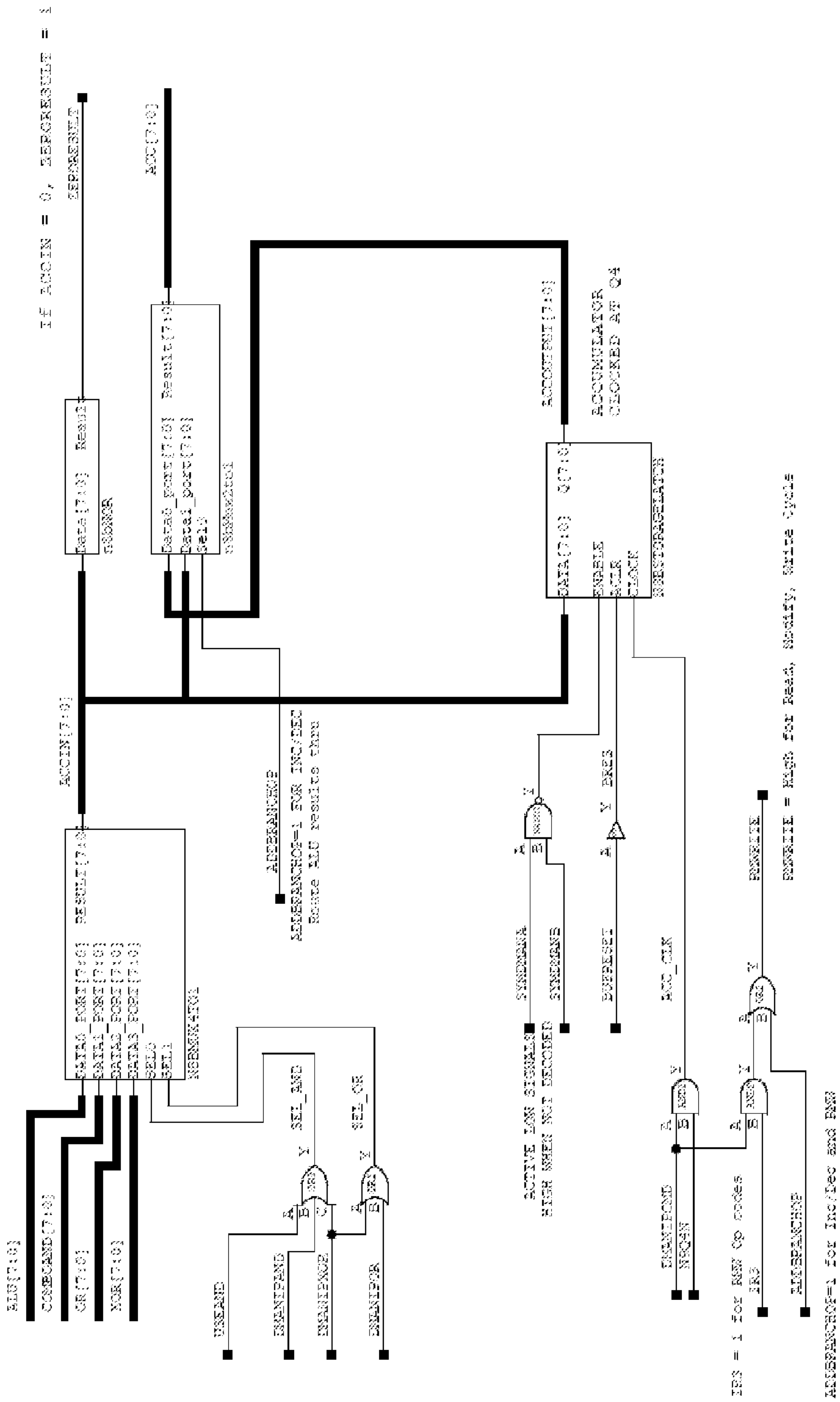


FIG. 28

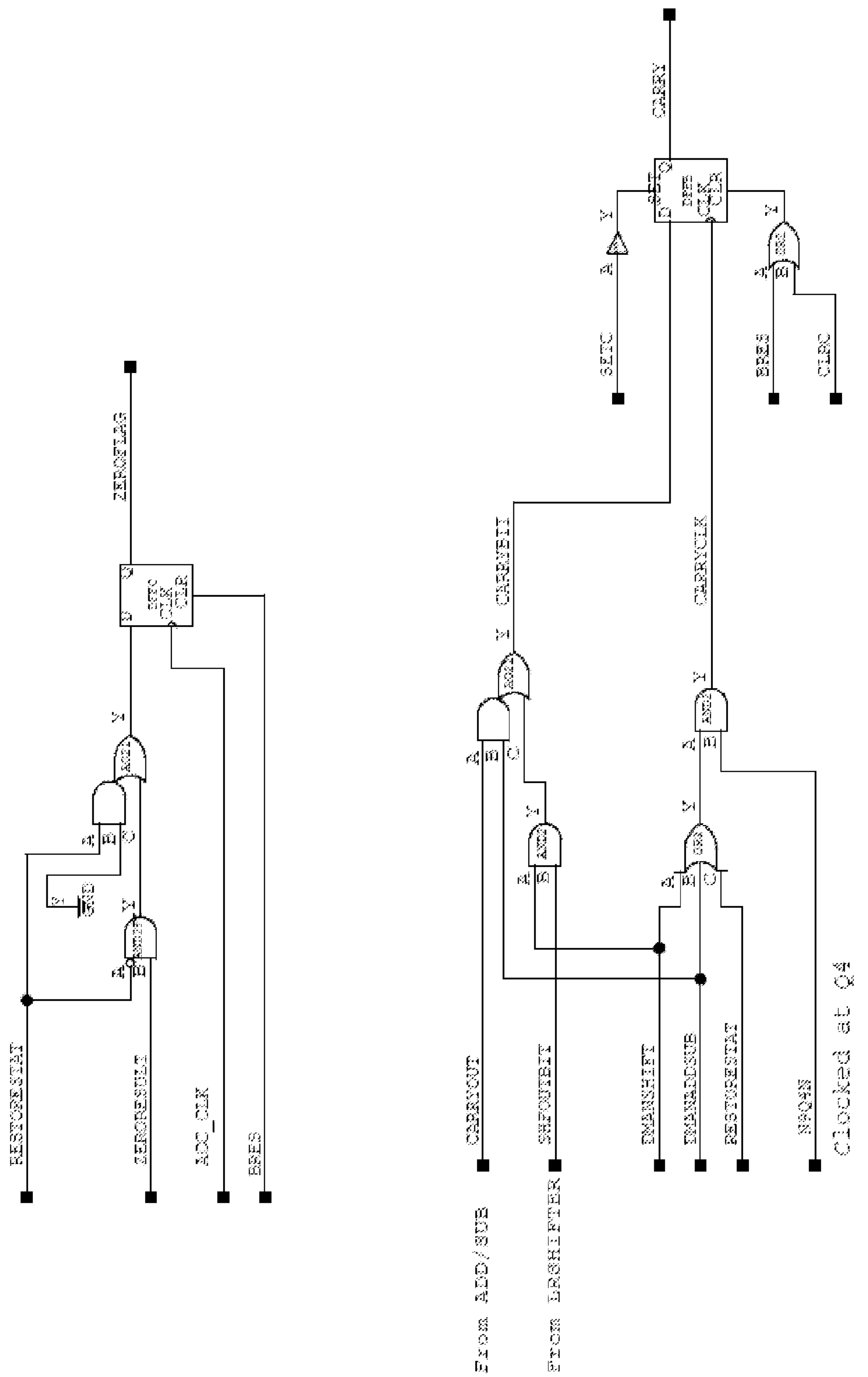

ZEROFLAG
RESTORESTAT
ACC_CLK
BRES
CARRY
SETC
BRES
CLRC
CARRYBIT
CARRYCLK
CARRYOUT
RESTORESTAT
Clocked at Q4
From ADD/SUB
From LRSHIFTER

FIG. 29

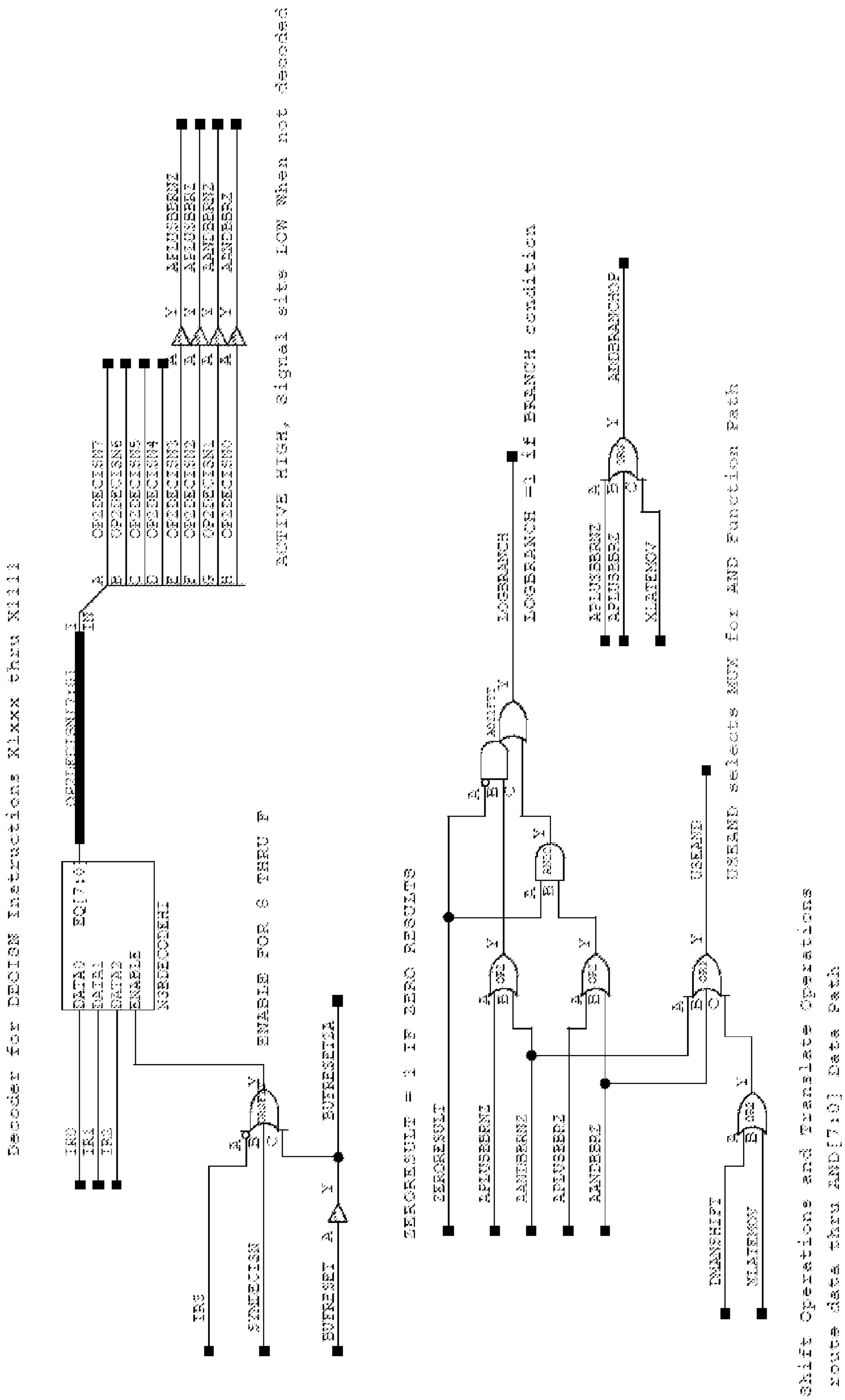

Decoder for DECISN Instructions X1xxx thru X1111
ACTIVE HIGH, signal sits LOW when not decoded
ENABLE FOR 8 THRU F
ZERORESULT = 1 IF ZERO RESULTS
LOGBRANCH =1 if BRANCH condition
USEAND selects MUX for AND Function Path
Shift Operations and Translate Operations
route data thru AND[7:0] Data Path

FIG. 30

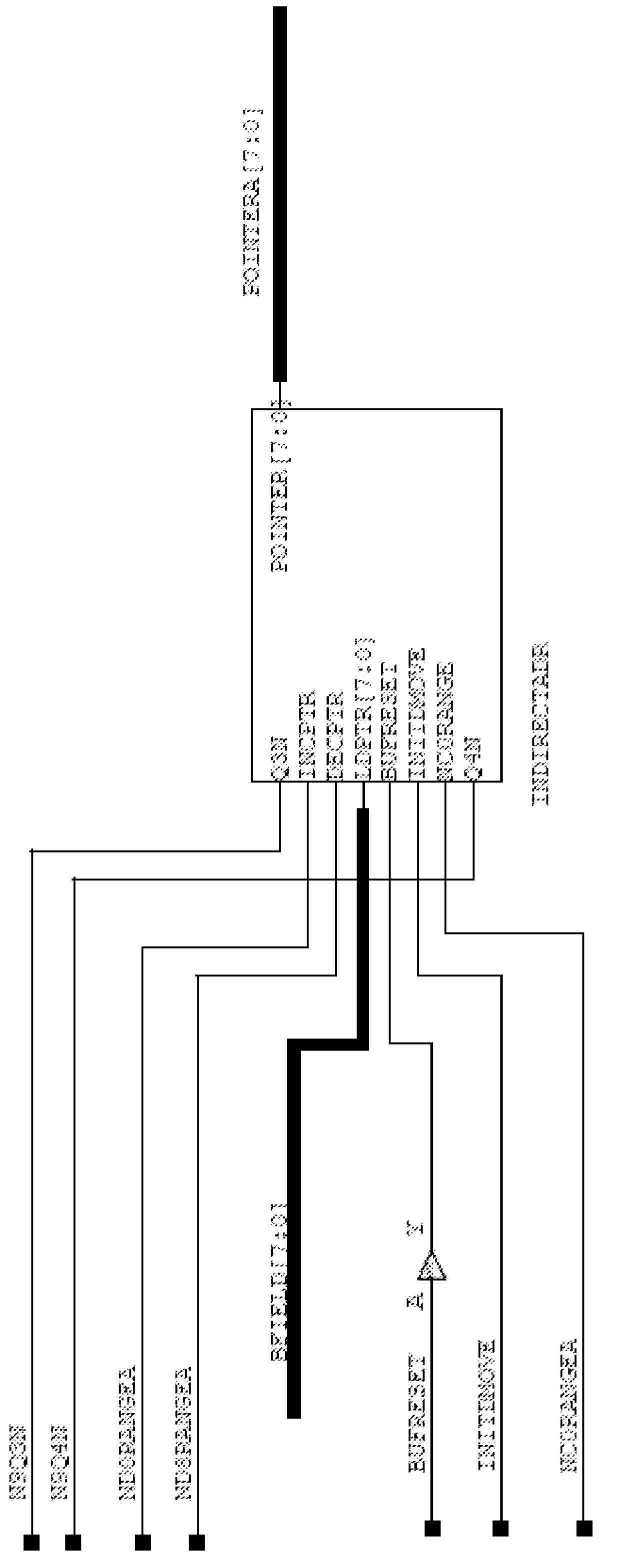

POINTERA[7:0]
POINTER[7:0]
INCPTR
DECPTR
LDPTR[7:0]
BUFRESET
INDIRECTADR
BUFRESET

FIG. 31

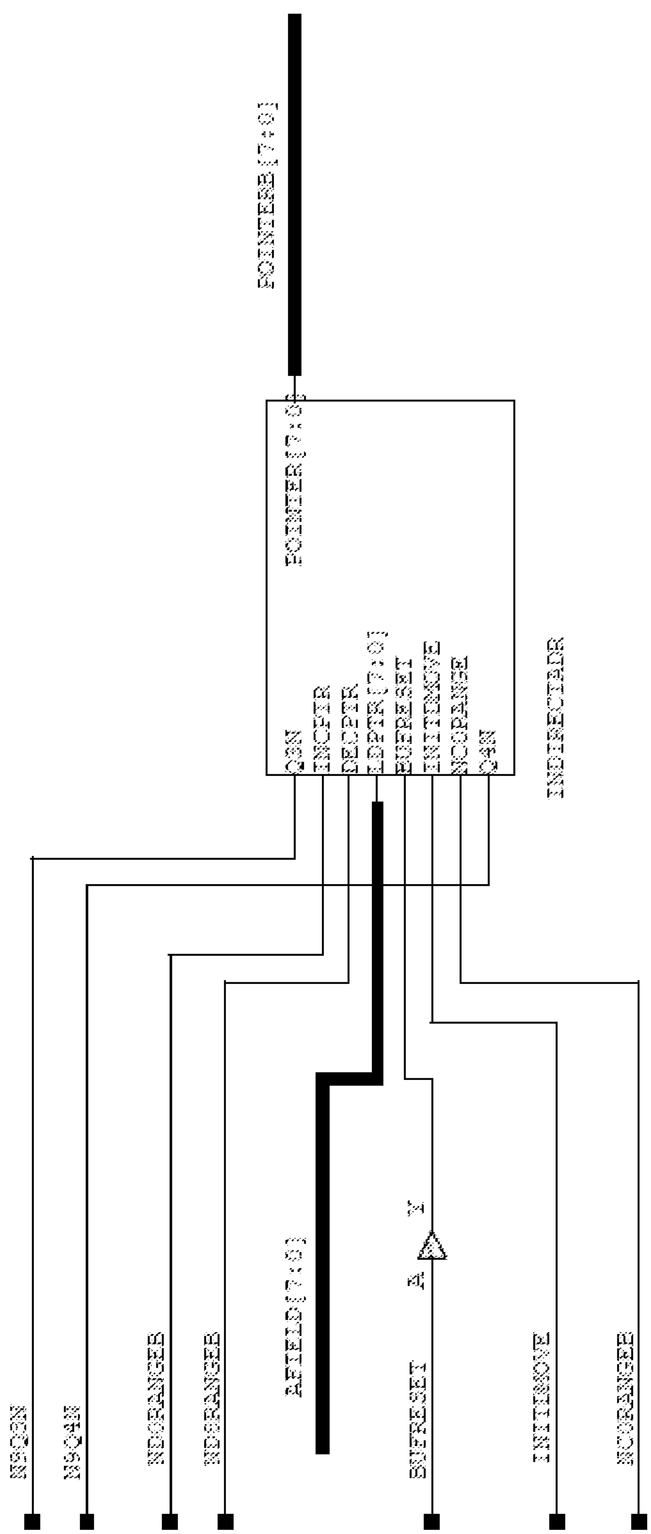


FIG. 32

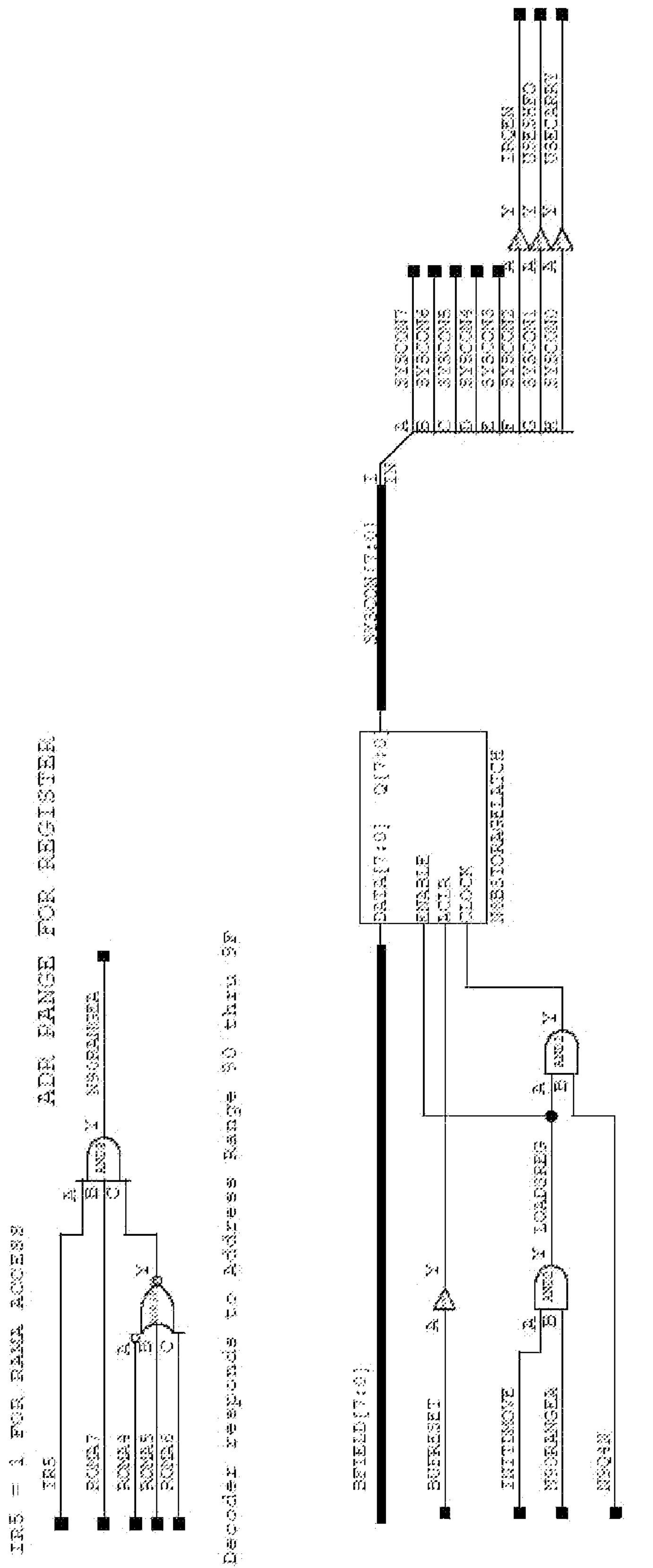
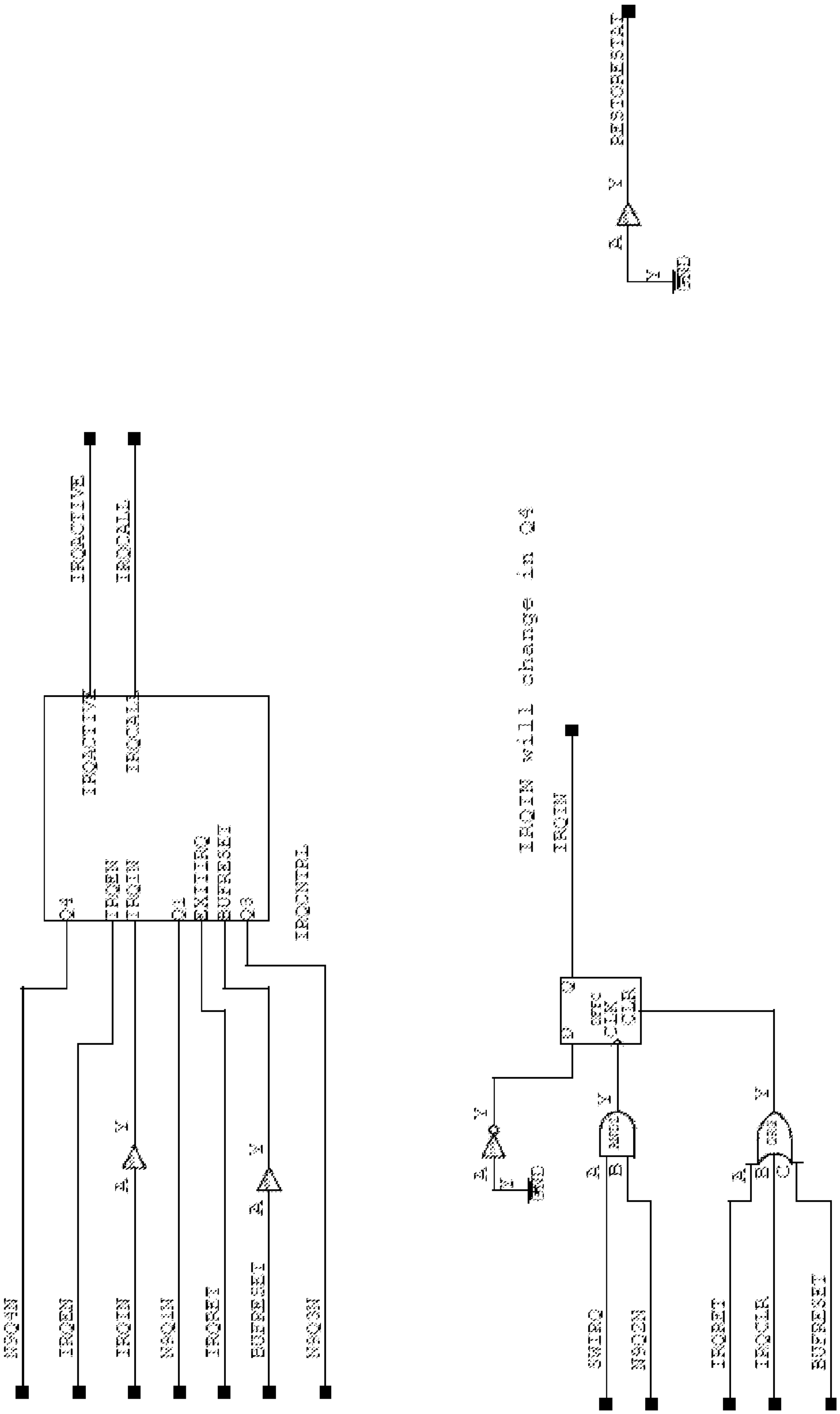

ADR RANGE FOR REGISTER
IR5 = 1 FOR RAMA ACCESS
Decoder responds to Address Range 90 thru 9F
SYSCON[7:0]
BYTELD[7:0]
DATA[7:0]
Q[7:0]
ENABLE
ACLR
CLOCK

FIG. 33



RESTOREINT
GND
IRQACTIVE
IRQCALL
IRQIN will change in Q4
IRQIN
Q4
IRQEN
IRQIN
Q1
EXITIRQ
BUFRESET
Q3
IRQINTRL
NSQ4N
IRQEN
IRQIN
NSQ1N
IRQRET
BUFRESET
NSQ3N
SWIRQ
NSQ2N
IRQRET
IRQCLR
BUFRESET

FIG. 34

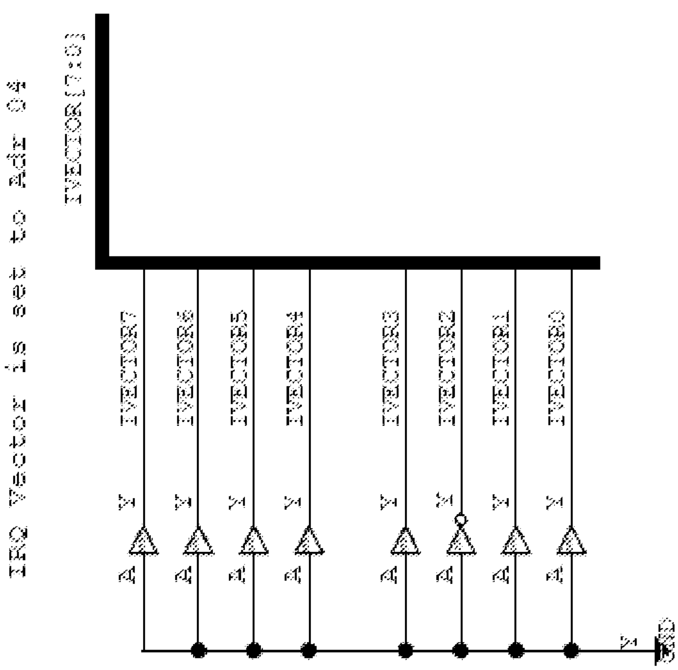

IRQ Vector is set to Adr 04
IVECTOR[7:0]
IVECTOR7
IVECTOR6
IVECTOR5
IVECTOR4
IVECTOR3
IVECTOR2
IVECTOR1
IVECTOR0
GND

FIG. 35

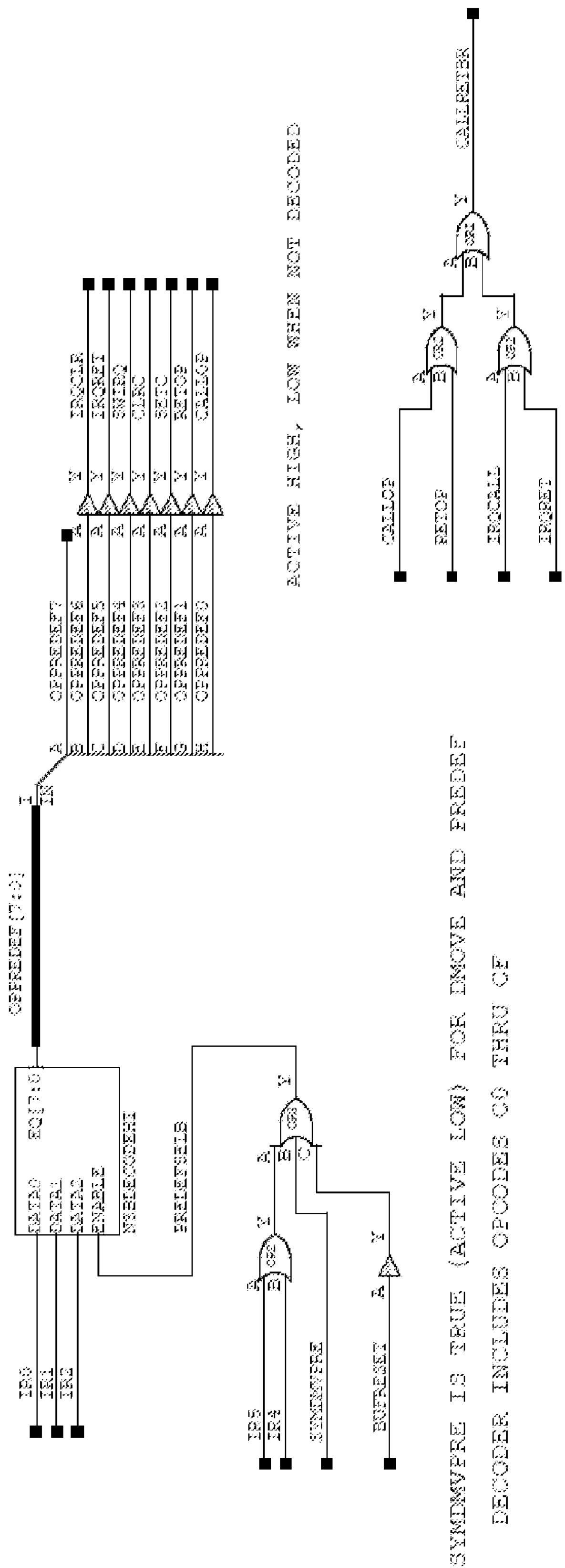

ACTIVE HIGH, LOW WHEN NOT DECODED
SYNDMVPRE IS TRUE (ACTIVE LOW) FOR DMOVE AND PREDEF
DECODER INCLUDES OPCODES C0 THRU CF

FIG. 36

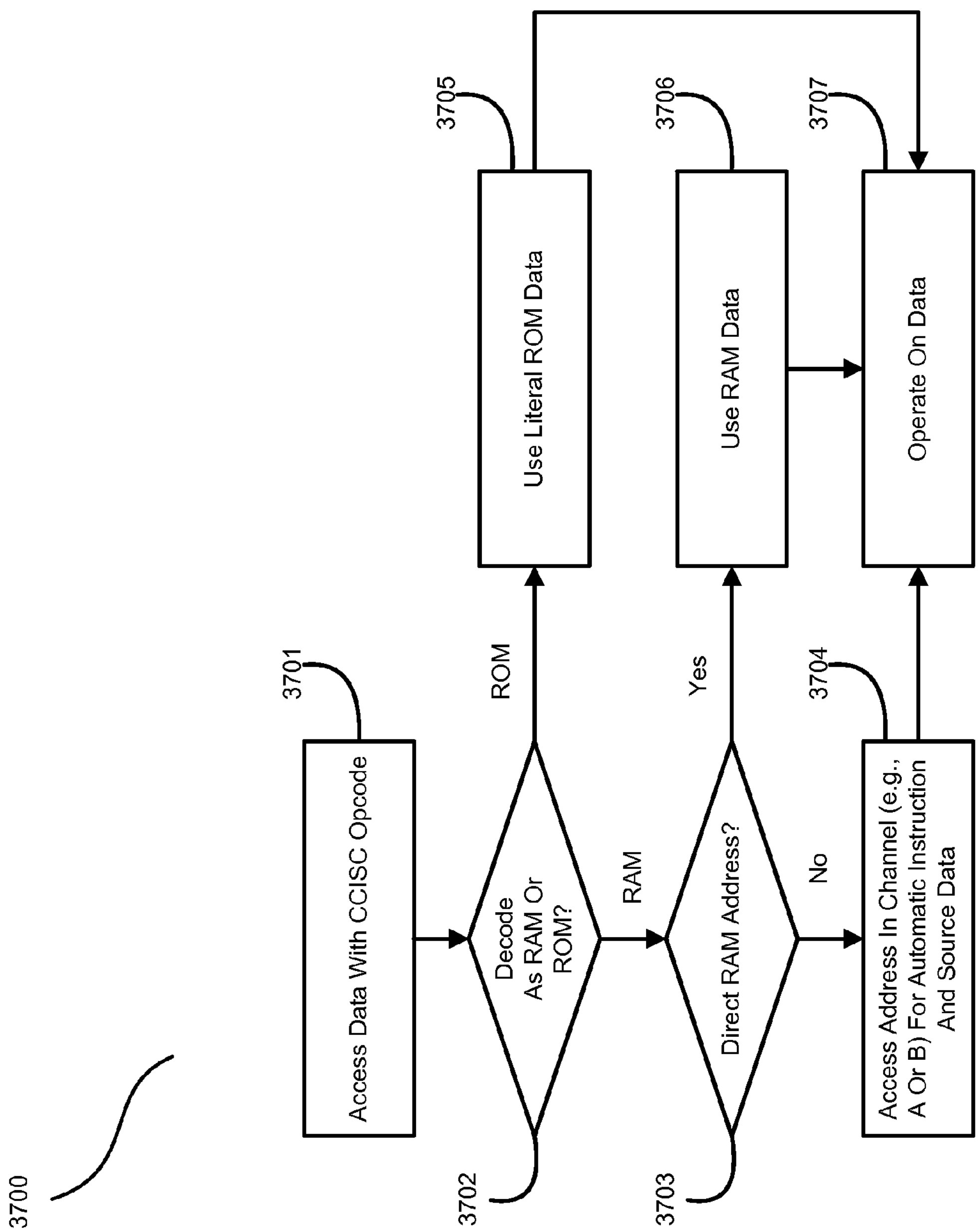

3700
3701
Access Data With CCISC Opcode
3702
Decode As RAM Or ROM?
ROM
RAM
3703
Direct RAM Address?
Yes
No
3704
Access Address In Channel (e.g., A Or B) For Automatic Instruction And Source Data
3705
Use Literal ROM Data
3706
Use RAM Data
3707
Operate On Data

FIG. 37

FLOWCHART COMPILER FOR A COMPOUND COMPLEX INSTRUCTION SET COMPUTER (CCISC) PROCESSOR ARCHITECTURE

This patent application is related to commonly owned and co-pending U.S. patent application Ser. No. 13/746,239 filed herewith and entitled "COMPOUND COMPLEX INSTRUCTION SET COMPUTER (CCISC) PROCESSOR ARCHITECTURE"), the entire contents of which are incorporated by reference.

FIELD OF THE INVENTION

The invention relates to software compilers, and more specifically to a flowchart compiler for generating executable code for a compound instruction set processor.

BACKGROUND

Computer architecture regards the basic structure of a computer from the standpoint of what exactly can be performed by code written for the computer. Ordinarily, architecture is defined by the number of registers in the Central Processing Unit (CPU), the logic operations performed by the Arithmetic and Logic Unit (ALU), etc. Usually, the architectural definition is expressed as an instruction set and its related elaboration. Some computer architectures have been founded on reduced instruction sets to provide performance advantages. Complex Instruction Set Computer (CISC) processors and Reduced Instruction Set Computing (RISC) processors are two examples of such.

RISC processors have an architecture based on simplified instructions capable of providing higher performance due to faster execution of each instruction. The general concept is that RISC processors use a small, highly-optimized set of instructions, rather than a more specialized set of instructions often found in other types of architectures. RISC processors use a load/store architecture that allows memory to be accessed by load and store operations with all values for an operation being loaded from memory and present in registers. After the operation, the result is stored back to memory.

CISC processors, on the other hand, are generally characterized as having a larger number of instructions in their instruction set, often including memory-to-memory instructions with complex memory accessing modes. The instructions are usually of variable length, with simple instructions being only perhaps one byte in length with complex instructions being in the dozens of bytes in length. The size of an operand specifier generally depends upon the addressing mode, etc. The first byte of the operand specifier describes the addressing mode for that operand, while the opcode defines the number of operands. When the opcode itself is decoded, the total length of the instruction is not yet known to the processor because the operand specifiers have not yet been decoded.

One advantage of CISC processors lies in the source code that generally result in more work being done by the processor for each line of code. But, this comes at the expense of execution time, more so when pipelining of instruction execution is necessary to achieve desired performance levels. The advantage of RISC processors, therefore, lies in the speed of execution of code, although less is accomplished by each line of code.

During the software development process, a developer will often "sketch out" the higher level details about how the software will operate utilizing flowcharts. These higher level

details are an abstraction of the concepts of the software, such that the developer may concentrate on the overall flow and functionality of the software rather than the implementation details for any particular development language or processor. The developer will then attempt to translate the software concepts captured in the flowcharts into a higher level text based programming language, such as C, C++, Assembly Language, Visual Basic, etc. The text based programming language is then compiled into executable software for execution by the processor.

SUMMARY

Systems and methods herein provide for a compiler to create executable programs for a compound instruction based processor directly from flowcharts. Compound CISC, or "CCISC", overcomes the problems and tradeoffs associated with RISC and CISC processors by executing complex instructions, or opcodes, in a single clock cycle without incurring penalties for using any combination of advanced addressing modes. The flowchart compiler reads flowchart diagram files, and interprets the flowchart symbols and arrows connecting the symbols that are defined in the diagram files to generate executable code for a CCISC processor. The flowchart symbols may be grouped into different computing categories, such as decision making, data manipulation, data movement, predefined functions, function entry/exit, etc., such that the flowchart compiler may readily map the flowchart symbols to CCISC assembly language instructions or combinations of CCISC assembly language instructions. Because flowchart diagrams are easier to read and understand than a high level text based language, both programmers and non-programmers alike may readily recognize the software concepts that are implemented by the flowcharts. Further, flowcharts are a self-documenting graphical implementation of the developed software, which helps programmers understand pre-existing programs in less time than parsing though a text based programming language.

Generally, the CCISC assembly language is broken into four main types of opcodes which may be mapped to different computing categories of flowchart symbols. Decision opcodes may map to a flowchart decision category, and compare two data values and conditionally branch to one branch target address while executing in a single clock cycle. The decision opcodes use a separate compare engine in a CCISC processor and do not employ an accumulator. Data manipulation, or "DMANIP", opcodes may map to a flowchart data manipulation category and include arithmetic and logical operations that change data in some way. DMANIP opcodes can use one, two, or three data values, and can perform its operations directly from memory, again without employing an accumulator. By using two data values and one target address value, DMANIP opcodes can operate on two data values and store the results in a third location in a Random-access memory (RAM) bank. Data move, or "DMOV", opcodes may map to a flowchart data movement category and can move data in the processor using two data values. The DMOV is comparable to a Direct Memory Access (DMA) in that the processor does not use an accumulator. Other predefined opcodes may be mapped to a predefined function category, and may be used to cover other instructions such as CALL, RETURN, NOP, SLEEP, SETC, CLRC, etc. Additionally, some predefined instructions may be fixed within certain address locations in memory so as to be address activated.

In one embodiment, a system is disclosed that includes an interface and a flowchart compiler. The interface is operable to receive one or more flowchart diagram files that represent a computer program for a Compound CISC (CCISC) processor. The flowchart compiler is operable to identify a flowchart symbol in the one or more flowchart diagram files, to identify a computing category for the flowchart symbol, and to generate one or more CCISC instructions based on the computing category for execution by the CCISC processor, where the one or more CCISC instructions are operable to direct the CCISC processor to access and operate on at least two data values in a multi-channel memory during the same clock cycle.

The various embodiments disclosed herein may be implemented in a variety of ways as a matter of design choice. For example, the embodiments may take the form of computer hardware, software, firmware, or combinations thereof. In one embodiment, a method is operable within the processor system to perform the functionality described herein. In another embodiment, a computer readable medium is operable to store software instructions that are operable to implement the various steps of the method. For example, the executable program may be stored on a computer readable medium so as to direct the processor to retrieve the opcodes from the program memory and operate in the manner described above. Other exemplary embodiments may be described below.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** is a block diagram of an exemplary system for generating CCISC instructions from flowcharts.

FIG. **2** is a flowchart of an exemplary method operable with the system of FIG. **1**.

FIG. **3** illustrates an exemplary embodiment of program memory operable with the CCISC processor of FIG. **1**.

FIG. **4** illustrates an exemplary multichannel memory operable with the CCISC processor of FIG. **1**.

FIG. **5** illustrates exemplary predefined instructions within the multichannel memory that are address range activated.

FIG. **6** is a block diagram of another exemplary embodiment of the CCISC processor system.

FIGS. **7-36** illustrate various exemplary logic diagrams used to implement components of the CCISC processor system of FIG. **6**.

FIG. **37** is a flowchart illustrating another embodiment of the CCISC opcode processing.

DETAILED DESCRIPTION OF THE DRAWINGS

The figures and the following description illustrate specific exemplary embodiments of the invention. It will thus be appreciated that those skilled in the art will be able to devise various arrangements that, although not explicitly described or shown herein, embody the principles of the invention and are included within the scope of the invention. Furthermore, any examples described herein are intended to aid in understanding the principles of the invention, and are to be construed as being without limitation to such specifically recited examples and conditions. As a result, the invention is not limited to the specific embodiments or examples described below.

FIG. **1** is a block diagram of an exemplary system **150** for generating CCISC instructions from flowcharts. The system **150** includes an interface **152** and a flowchart compiler **153**. The interface **152** comprises any system, component, or device that is operable to receive one or more flowchart diagram file(s) **151** that represent a computer program for a CCISC processor as one or more flowcharts. The flowchart compiler **153** comprises any system, component, or device that is operable to generate CCISC instructions **154** based on the flowchart(s) embodied in the flowchart diagram file(s) **151**.

Discussion of the system **150** is now directed to the exemplary flowchart **200** illustrated in FIG. **2**. In FIG. **2**, the interface **152** receives one or more flow chart diagram files **151** that represent a computer program for a CCISC processor in a flowchart format, in the process element **201**. Generally, flowcharts include various symbols that describe high level representations of a process, such as a computer program. The flowcharts also include arrows that define how the symbols are logically connected together. In other words, the arrows define the flow of the overall process defined by the flowchart symbols. The flowchart compiler **153** identifies a flow chart symbol in the flow chart diagram files, in the process element **202**. The flowchart compiler **153** then identifies a computing category for the flowchart symbol, in the process element **203**. The flowchart symbols may be grouped into different computing categories, such as decision making operations, data manipulation operations, data movement operations, pre-defined function operations (e.g., processor or hardware specific commands or operations), initiate/terminate labels (e.g., a representation of the beginning or end of a subroutine), manual input operations (e.g., a representation of data definitions, constant values, and/or absolute address locations to constant values defined in the computer program. Flowchart compiler **153** may readily map the flowchart symbols to CCISC instructions (i.e., assembly language instructions) **154** or combinations of CCISC instructions **154**.

The flowchart compiler **153** generates one or more CCISC instructions **154** based on the computing category identified for the flow chart symbol, in the process element **204**. In this embodiment, the CCISC instructions **154** generated by the flowchart compiler **153** direct a CCISC processor (not shown in FIG. **1**) to access and operate on at least two data values in a multi-channel memory (not shown in FIG. **1**) during the same clock cycle.

For decision making operations, the flowchart compiler **153** may generate one or more CCISC instructions **154** to compare the two data values to branch to an address in program memory. For data manipulation operations, the flowchart compiler **153** may generate one or more CCISC instructions **154** to change at least one of the data values in the multichannel memory. For data movement operations, the flowchart compiler **153** may generate one or more CCISC instructions **154** to move one data value from a first address to a second address in the multichannel memory. Examples of each are shown and described in greater detail below.

For example, if a CCISC processor, when executing CCISC instructions **154**, determines that the instruction is a decision instruction, DMANIP instruction, or a DMOV instruction, then the CCISC processor accesses a first data value from a channel A of the multichannel memory and a second data value from a channel B of the multichannel memory. The CCISC processor then operates on the first and second data values in their respective locations of the multichannel memory based on the CCISC instruction. That is, the CCISC processor does not utilize an accumulator or other register to store the data values from the multichannel memory such that the data values may be operated on and then returned to the multichannel memory. Rather, the CCISC processor operates on the data values directly within the multichannel memory.

In some embodiments, the flowchart compiler **153** identifies text within the flowchart symbol that defines computing operations for the computing category. The flowchart compiler **153** then generates the CCISC instructions **154** based on the text. For example, the text may specify VAR3=VAR1+4 within a flowchart symbol residing in a data manipulation category. In this case, the text conveys the specifics of the computing operation to perform. In general, the rules for data entry into a flowchart symbol may follow that of the C programming language and many Assembly Languages. Upper and lower case alpha characters may specify different variables. Further, different punctuation marks (e.g., +, −, =, <, >, etc.,) may be used to denote mathematical operations.

Arrows generally illustrate program flow and execution order within a flowchart program from one flowchart symbol to another flowchart symbol. Using flowchart symbols and arrows allows the programmer to readily recognize the flow of the computer program without resorting to locating and matching label names or other types of flow identifiers within the typical text based programming languages.

To support the equivalent of labels, a flowchart program may utilize a specialized terminal symbol. The terminal symbol may have different meanings depending on whether an arrow enters or exits the terminal symbol. A terminal symbol with an arrow that exits to another non-terminal symbol may indicate the beginning of a function or other subroutine. A terminal symbol with an arrow that enters from another non-terminal symbol may indicate an exit point for the subroutine. Other arrow configurations may illustrate branching decision paths within the flowchart program.

To provide more context, if the CCISC processor determines that the opcode is not a decision opcode, a DMANIP opcode, or a DMOV opcode, then the CCISC processor determines whether the opcode is located in channel A of the multichannel memory. If the opcode is located in channel A of the multichannel memory, then the CCISC processor accesses the address where the instruction is located in channel A for automatic instruction or direction. Depending on an address read from the A Field of the opcode, the CCISC processor addresses Channel A of the multichannel memory. If the address falls in a specific range of addresses, it causes automatic instructions to initiate, in addition to the Decision, DMOV, DMANIP operations implemented by their respective opcodes.

For example, the CCISC processor may process the opcode and determine that the opcode is actually a particular address in channel A of the multichannel memory. From there, the CCISC processor may access that address in channel A of the multichannel memory where a predefined instruction is located. The address itself causes the automatic instruction to operate with no further fetch of an opcode being necessary. Examples of such predefined/automatic instruction include pointer increments, status checks, etc. Thus, the opcode may be "address actuated" and that the CCISC processor is automatically directed to the address location in the channel A of the multichannel memory for automatic instruction. If the CCISC processor determines that the opcode is not located in channel A of the multichannel memory , then the CCISC processor accesses an address in channel B where the instruction is located for similar automatic instruction or direction. In either case, after the address actuated instruction is executed, the CCISC processor determines whether the program is at its end again returning to process element if it is not and progressing to process element if so.

To summarize, the decision, DMOVE, DMANIP, and predefined opcodes read the A and B data sources. Depending on the automatic instruction to determine where the data is located, the opcodes provide additional operations, such as pointer manipulation. And, there are two sets of "address actuated" circuits that provide automatic instructions for both the A and B data sources.

FIG. **3** illustrates an exemplary embodiment of a program memory **101** operable with a CCISC processor (see e.g., FIG. **6**). In this embodiment, the program memory **101** is a 32-bit wide memory having addresses of 0×0000 through 0×FFFF (i.e., 65,536 addresses). Each 32-bit word in the program memory **101** is subdivided into four 8-bit fields, comprising the A field (bits **31**-**24**), the B field (bits **23**-**16**), the opcode field (bits **15**-**8**) and the link field (bits **7**-**0**). The A field provides the data or address of channel A in A multichannel memory **103** (FIG. **6**) during an instruction fetch for the current CCISC instruction of the executable program being processed by the CCISC processor. The B field similarly provides the data or address of channel B in the multichannel memory **103** during the instruction fetch. The opcode field provides the operation code during the instruction fetch. The link field provides address data during the instruction fetch as well as target addresses within the multichannel memory **103** for a DMANIP opcode or a decision opcode. The link field may also be used for various predefined opcodes that are address actuated in the multi-channel memory **103**.

FIG. **4** illustrates the multichannel memory **103** operable with the CCISC processor. For the purposes of illustration, the multichannel memory **103** is illustrated as two separate channels **103**A and **103**B representing the channels A and B described above. In this embodiment, channels A **103**A and B **103**B have address ranges from 0×00 to 0×FF (i.e., 256 addresses). Each channel **103**A and **103**B is designated with two sections, data values **110** for addresses 0×7F and below, and address range activated opcodes for addresses 0×80 and above. The data values **110**A and **110**B may be used to store any data values required by the executable program, much like any other processing system. One difference between prior art processing systems and the present processing system is that the CCISC processor operates on the data values directly within the channels **103**A and **103**B of the multichannel memory **103**. In the opcode sections **111**A and **111**B, the channels **103**A and **103**B store their respective predefined opcodes. An example of such is illustrated in FIG. **5**.

FIG. **5** illustrates exemplary predefined instructions within the multichannel memory **103** that are address range activated. For example, when the CCISC processor is directed to access address 0×FF in channel **103**A of the multichannel memory **103**, the CCISC processor is automatically directed to perform a pointer decrement (PtrA−−) within the channel **103**A. Similarly, when the CCISC processor is directed to access address 0×FF in channel **103**B of the multichannel memory **103**, the CCISC processor is automatically directed to perform a pointer decrement (PtrB−−) within the channel **103**B. Other examples of predefined opcodes include STATUS, CALL, RETURN, NOP, SLEEP, SETC, CLRC, pointer increments, such as Ptr++ and ++Ptr, and indirect address accesses through pointers (e.g., PtrA and PtrB). Of course, the invention is not intended to be limited to any particular address allocation or type of opcode being configured within the channels **103**A and **103**B of the multichannel memory **103**.

FIG. **6** is a block diagram of another exemplary embodiment of the CCISC processor system **100**. The processor system **100** includes the program memory **101** of FIG. **1** as well as the multichannel memory **103** configured as a dual channel RAM bank adding channels **303**A and **303**B and a RAM Address Generation Unit (RAM AGU) **312**. The CCISC processor is configured from the reset and program boot loader **301** (boot loader **301**), a clock **302**, a data router

304, an opcode decoder and CPU controller 305, an AGU program counter and branch address vector module 306 (AGU), a stack and interrupt control module 307, an algorithmic logic unit (ALU) 308, a compare engine 309, and a register bank input/output (I/O) 310. The CCISC processor may be optionally coupled to peripherals 311, such as other devices or computing modules.

The clock 302 provides timing control and uses an input clock and an input clock shifted by 90 to create four quadrature clocks without clock multiplying. The lower speed input clocks save power and creates quadrature clocks Q1, Q2, Q3, Q4, as illustrated in FIG. 7 with references numbers N9Q1N-N9Q4N.

The boot loader 301 is coupled to the program memory 101 to initiate operations of the CCISC processor. The boot loader 301 controls the reset of the CCISC processor and each of its control registers. A "RESET" opcode provides "power-up and time-out counters" that hold the CCSIC processor in reset while a power supply (not shown) stabilizes. The RESET clears internal processor control registers and sets default power-up conditions. In addition to being coupled to the program memory 101, the boot loader 301 may be coupled to external memory, such as a serial Electrically Erasable Programmable Read-Only Memory (EEPROM), through a serial peripheral interface (SPI) to boot the CCISC processor, although the processor system 100 may be configured with internal non-volatile memory.

Once the boot loader 301 is initialized and activated, the boot loader 301 may reset the program counter of the AGU 306 to 0×0000. The boot loader 301 then reads the external memory and writes the data into the program memory 101. The boot loader 301 loads the program memory 101 with executable code and then passes control to the AGU 306 which starts at the address 0×0000.

The boot loader 301 may be configured with a variety of modules, such as a shift register generator/control 800 illustrated in FIG. 8, an output shift register 900 illustrated in FIG. 9, an input shift register 1000 illustrated in FIG. 10, and a boot loader control 1100 illustrated in FIG. 11. The shift register generator/control 800 interfaces to the program memory 101 and includes a clock generator 801 that provides the clock for the SPI interface. A clock inter-burst delay counter 802 provides for a delay between bursts of SPI clocks. The initialize and lockout feed forward module 803 establishes the SPI state machine for reading the external memory contents before passing control to the CCISC processor. The clock counter 804, in general, counts every 4 external data bytes for writing into the program memory 101. The output and input shift registers of and state machines to communicate with a SPI interface. The output shift register 900 of FIG. 9 uses a SPI "Data Shift Out" to output data from the external memory for serial communications, but is generally not needed when booting is performed from the program memory 101.

The boot loader control 1100 enables the start state of the boot loader 301 from RESET and detects the end state. A boot loader enable module 1101 enables the state of the boot loader 301 and disables the state of the boot loader 301 when passing control to the AGU 306 and CCISC processor in general. Also, at the beginning of the state of the boot loader 301, all instruction execution is generally locked out. A power up enable module 1102 is activated by the external RESET signal and provides a power up and reset of the loader 301 and the CCISC processor in general. A stabilizer counter 1103 provides delay for voltage stabilization purposes at RESET.

A state machine counter 1200 counts the number of bytes loaded from the external memory and signals when to stop downloading (i.e., TOTAL BYTES COUNTER). The end of the download state is also provided by the RESET. The "EE CHIP SELECT" drives the selection from external memory and the program memory 101. A logical high between third and fourth clock cycles enables a continuous read operation from the external memory.

The AGU 306 addresses the program memory 101 for fetching the CCISC opcodes to execute. One example of the AGU 306 is illustrated in FIG. 13. The program counter 1307 generates addresses for the program memory 101 and can be altered by a branch type of instruction (e.g., branch addresses, CALL and RETURN subroutines, absolute addressing, and±relative branch addressing) or by address vector (e.g., interrupt vector addressing). In one embodiment, the program counter 1307 address of the AGU 306 is 16 bits wide, having 64K address range, starting at 0×0000 after RESET, and generally counting up from there. A branch control input module 1308 is used for loading branch addresses into the AGU 306. A "4-cycle" control module supports the boot loader 301 to write data from the external memory to be program memory 101 on a byte to byte basis.

The stack and interrupt control module 307 is coupled to the AGU 306 to provide an address stack for storing return addresses during function calls or address vector events, such as interrupts. The stack and interrupt control module 307 includes the address stack and muxes for use with the AGU 306 and the program counter 1307 thereof. In one embodiment, the stack is a 128×16 address stack with 128 16-bit entries having a self-contained stack control and a return address with a plus or minus 7 bit offset. An example of the stack and interrupt control module 307 is illustrated in FIG. 14, including interrupt control block for containing the interrupt enable/disable control, an interrupt edge synchronizer, and an interrupt vector address. The stack and interrupt control module 307 activates the interrupt and loads the interrupt address vector into the AGU 306 upon receiving an interrupt request. Upon executing a RETURN from interrupt, the return address is loaded from the stack and the interrupt is turned off. For example, the subroutine RETURN may restore a return address plus 1 or some other offset value from the address stack.

The stack and interrupt control module 307 also includes an interrupt state machine that senses and synchronizes interrupt signals restores ACC (i.e., Accumulator) and STATUS. Data upon an existing interrupt mode, and is relatively fast with 1½ to 2½ clock interrupt latency. Additionally, the stack and interrupt control module 307 controls RAM A 303A and RAM B 303B for automatic saving ACC and STATUS data during interrupts. An example of such as illustrated in FIG. 14. The stack and interrupt control module 307 also supports the subroutine CALL while saving the return address on the stack.

As mentioned, the program memory 101 may be 32 bits wide to accommodate 32 bit CCISC opcodes. To provide the opcodes to the processor, the program memory 101 may be configured with an opcode ROM and link ROM with write data for these ROMs coming from the boot loader 301 and the input shift register 1000 thereof. An example of such is illustrated in FIG. 15. To decode the opcodes, the CCISC processor employs the opcode decoder and CPU controller 305. The decoder/CPU controller 305 block reads opcode fields from the program memory 101 to decode current CCISC instructions. Decoder/CPU controller 305 also controls basic data routing for the A and B data fields of the CCISC instructions (i.e., for the RAMs 303A and 303B).

FIG. 16 illustrates an instruction register 1601 which holds a current opcode. The decoder/CPU controller 305 is the module of the CCISC processor that decodes the four types of

CCISC instructions (i.e., decision opcodes, DMANIP opcodes, DMOV opcodes, and the predefined opcodes such as CALL, RETURN, NOP, SLEEP, SETC, CLRC, etc.). The decoder/CPU controller **305** can enable and disable all CCISC instructions within the CCISC processor. FIG. **17** illustrates a decision instruction decoder module **1701** of the decoder/CPU controller **305**. The decision instruction decoder module provides decisions for <, =, >, <=, !=, and >= comparison operations. The decision instruction decoder module **1701** also decodes "branch always" decision opcodes and provides for branch enable flip-flop.

The data router **304** provides multiple data sources for the A and B data fields of the CCISC instructions. Internal A and B data buses may each be driven by a 4-input, 8-bit data mux to provide four data sources for the A and B data fields for each of the CCISC instructions. The data router **304** combines data routing with advanced addressing modes to provide the CCISC processor with many data source possibilities for each of the A and B data fields. The data router **304** multiplexes data sources for the A and B data fields (e.g., ROM A and RAM A of the program memory **101** for the A data field and ROM B, RAM B, and system buffer other program memory **101** for the B data field. Data muxes of the data router **304** provide for direct memory access to the RAMs **303**A and **303**B and may even swap data between the RAMs **303**A and **303**B. Write data for these data fields may be delivered by the loader **301** and the input shift register **1000** thereof. An example of the multiplexing by the data router **304** is illustrated in FIG. **18**.

The link bus **321**, A bus **322**, and the B bus **323** are internal 8-bit address buses coupling RAMs **303**A **303**B with various modules of the CCISC processor. It should be noted that this is a significant difference from other processors particularly 8-bit processors. More specifically, the A bus **322** is coupled to the RAM **303**A and the B bus **323** is coupled to the RAM **303**B such that the RAMs **303**A and **303**B provide a dual scratchpad for the CCISC processor. The output from the data router **304** drives the A bus **322** and the B bus **323** to support internal compound data. The link field of the link bus **321** is driven from the program memory **101** to provide provides single cycle writes and branches. The link bus **321** can also be used for addressing offset accesses.

The RAM AGU **312** provides complex addressing and intelligent memory functions. The RAM AGU **312** provides independent and simultaneous use of two different addressing modes for each CCISC construction. For example, the RAM AGU **312** may be configured to independently access each of the RAMs **303**A and **303**B. The RAM AGU **312** also provides indirect pointer and pre/post use pointer manipulation as discussed above. In one embodiment, pointer registers are implemented with up-and-down counters, but can be implemented using an adder for different increment/decrement offsets. The independent pointer manipulation by the RAM AGU **312** reduces clock cycles for the CCISC processor. Predefined pointers within the RAMs **303**A and **303**B also offload execution from the CCISC processor by pushing such processing onto the RAM AGU **312**. As mentioned above, the RAMs **303**A and **303**B may be configured with 256 8-bit locations with the lower 128 locations being directly addressable and the upper 128 locations being indirectly addressable. However, the memory can be extended and/or the locations may be configured as a matter of design choice.

The ALU **308** supports DMANIP CCISC instructions of the CCISC processor. The ALU **308** provides functionality such as add, subtract, shift/rotate left, shift/rotate right, logical AND, logical OR, and logical XOR as well as carry and zero flags. Bit testing decision instructions use the ALU to perform a logical AND with the contents of a data field and the bit mask to test for one or more bits being set or cleared.

The ALU **308** is operable to test for several bits at once in a single clock cycle. For example, if the results of an AND is equal to 0, then all of the bits are cleared. If the results of the AND is not equal to 0, then at least one bit is set. The ALU **308** also uses auto count and test decision instructions to perform an ADD with the contents of a data field and 0×01 or 0×FF (negative 1) to support auto-count modes and checking for counter expiration. For example, if the result of an increment/decrement is equal to 0, then the count is rolled over to 0. If, however, the result of an increment/decrement is not equal to 0, then the count is not rolled over to 0. Outputs of the ALU **308** are routed back to the A and B field data muxes describes above. This makes the results of ALU **308** functions available to either the A or B fields for use in CCISC instructions.

The compare engine **309** accesses data of the A and B fields such that it can compare any two data values from multiple data sources in a single clock cycle. An example of the compare engine **309** is illustrated in FIG. **19**. As shown in FIG. **19**, the compare engine **309** provides for comparison operations without the use of an accumulator. Such reduces logic gates by not running compares through the ALU **308**. This also reduces clock cycles by using single cycle comparisons for all possible data source. A and B field data latches hold data read from A and B field buses **322** and **323**, respectively, to the compare engine **309**. Thus, the compare engine **309** is operable to connect the A and B field buses **322** and **323** to various components within the CCISC processor including onboard peripherals **311** and their various inputs and outputs. In one embodiment, the compare engine **309** uses and 8-bit magnitude comparator to provide results of the six standard comparisons: <, =, >, <=, !=, and >= of the A and B field data.

FIG. **20** illustrates the multiplexing of the A and B data fields. That is, various A field data and B field data sources are multiplexed onto the A and B field buses **322** and **323**. For example, A field data from an A field data source, such as ROM A data from the A field of the program memory **101**, RAM A data from RAM **303**A (including address accessible features), register data from multiple registers in RAM A address space, and accumulator data from an accumulator driven onto the A field bus **322**. Similarly, B field data sources may be obtained from ROM B data from B field of the program memory **101**, RAM B data from RAM **303**B (including address accessible features), a system buffer, and accumulator data from an accumulator driven onto the B field bus **323**.

FIG. **21** illustrates DMOVE instructions that are decoded, including the decoding of move and translate DMOVE instructions, and for moves relating to the loading of address range activated registers in the RAM **303**A and **303**B. The DMOVE provides data movement from any A sources to B destinations and from any B sources to A destinations. The CCISC instructions also provide for swapping A and B data via a SWAP instruction. In this embodiment, move and translate DMOVE instructions provide for A and B address ranges in upper and lower nibbles. The register bank I/O **310** provides a memory map of the RAM A address space, from 0×80 to 0×FF in one embodiment.

The A field, B field, and link field data/address fields connect to the register bank I/O **310** with the A field for being used for data transfer, the B field being used for data and addressing, and the link field being used for addressing and offset addressing. The register bank **310** may also provide a system buffer in the RAM **303**B address space. Examples of such are illustrated in FIGS. **22** and **23** for the A and B data fields, respectively, using the previously mentioned 256×8 RAM of which 128 are directly addressable and 128 are indirectly addressable.

FIGS. **24**-**36** illustrate various exemplary implementations of some of the functional aspects of the components described above. FIG. **24** illustrates how the above-mentioned address range activated features of the RAM **303**A and **303**B may be implemented. For example, the address range activated features of the RAM **303**A and **303**B address spaces may allow for independent decoding by the ALU **308** of the RAM address spaces thus providing for simultaneous and compound use of the address spaces.

FIG. **25** illustrates how the ALU **308** may decode DMA-NIP instructions including add, subtract, left shift/rotate, and right shift/rotate operations as well as the logical AND, OR, XOR operations. FIG. **26** illustrates how the ALU **308** may implement the left/shift rotate, the right shift/rotate, and the logical AND operations as well as the DMOVE operations (i.e., move and translate). The move and translate section includes the A and B data fields for the shift and DMOVE operations. In this embodiment, the ALU **308** is operable to translate upper and lower 4-bit hex nibbles from the A and B data fields into 8-bit ASCII using a single DMOVE instruction in one clock cycle. FIG. **27** illustrates an 8-bit adder/subtracter module that is operable to add, subtract, and provide logical OR and logical XOR operations.

FIG. **28** illustrates accumulator functionality of the ALU **308**. This functionality provides the ALU **308** with ability to route data mux results of the ALU **308** to various components including the RAMs **303**A and **303**B. FIG. **29** illustrates zero flag and carry flag functionality of the ALU **308** which may be used to indicate results of the operations of the ALU **308**.

FIG. **30** illustrates decoding of decision operations by the decoder/CPU controller **305**. The decoder/CPU controller **305** is operable to decode bit testing and auto-count testing of CCISCs decisions as well as including bit testing and auto-count testing branch control logic.

FIG. **31** illustrates an indirect pointer A register, part of the RAM AGU **312** that indexes the RAM A address space of the RAM **303**A. This functionality provides control of pre and post increment/decrement of pointer register values. FIG. **32** illustrates an indirect pointer B register, part of the RAM AGU **312** that indexes the RAM B address space of the RAM **303**B, also providing control of pre and post increment/decrement of pointer register values.

FIG. **33** illustrates system control register functionality of the CCISC processor, including decoding for the register, configuring CARRYs in various operations, controlling configurations of shifters, and enabling and disabling interrupts.

FIG. **34** illustrates an interrupt request state machine that synchronizes an interrupt event's edge with the clock **302**. The state machine also controls the active state of interrupts by activating an interrupt address vector and the AGU **306**. FIG. **35** illustrates an interrupt request address vector and mux that maintains the interrupt request vector address values. Generally a single vector is employed at any given time but other vector addresses can be muxed together.

FIG. **36** illustrates predefined decodes used by the decoder/CPU controller **305** to decode CALL, RETURN, Decodes Set CARRY, and Clear CARRY CCISC instructions. This functionality also provides for decoding software interrupt instructions and interrupt request returns for exiting interrupt modes of operation, as well as an enabling and disabling interrupts via an IRQCLR.

FIG. **37** is a flowchart **3700** illustrating another embodiment of the CCISC opcode processing. In this embodiment, the flowchart **3700** illustrates how a data source is determined for both the channel A data field and the channel B data field during the CCISC opcode decoding and prior to instruction execution. For the purposes of this embodiment, it is assumed that the CCISC processor received a CCISC opcode from program memory containing the A field, the B field, the opcode field and the link field. In this regard, the CCISC processor accesses data with the CCISC opcode in the process element **3701**. It should be noted that the flowchart **3700** is generally applicable to both the channel A field and the channel B field. The data source determination action for the A field and the B field are "compounded" and happen in parallel with opcode decoding. It should also be noted that each CCISC opcode contains two bit flags to indicate whether the A field data and the B field data represents "literal data" (referred to as ROM) or represents an "Address" (referred to as RAM).

The CCISC processor, based on opcode decoding, determines whether the data in the A and the B fields decoded as ROM (i.e., literal data) or RAM (i.e., addressable variable data), in the process element **3702**. If the data in the A field is decoded as ROM, the CCISC processor uses the A field data as a literal data constant, in the process element **3705**. The source of data for channel A is the data given in the A field from program memory **101**.

If the data in the A field is decoded as RAM, the CCISC processor determines that the A field data represents an address value and determines where the address is located, in the process element **3706** (e.g., between 0×00 to 0×7f or 0×80 to 0×ff). That is, the CCISC processor determines that the A field contains an address in the range of 0×00 to 0×7f, which addresses RAM. The CCISC processor uses the A field data as an address to RAM A memory. The source of data for channel A comes from an appropriate RAM **103** memory location (e.g., A or B).

If the data in the A field is decoded as an address in the range of 0×80 to 0×ff, the CCISC processor addresses the address range activated features discussed above, in the process element **3704**. The CCISC processor uses the A Field data as an address to address range activated features including multiple forms of indirect RAM **103** memory addressing and other automatic instructions, again depending on the source A or B. The CCISC processor operates on the data in the process element **3707**.

While the invention has been illustrated and described in detail in the drawings and foregoing description, such illustration and description is to be considered as exemplary and not restrictive in character. Certain embodiments described hereinabove may be combinable with other described embodiments and/or arranged in other ways. Accordingly, it should be understood that only the preferred embodiment and variants thereof have been shown and described and that all changes and modifications that come within the spirit of the invention are desired to be protected.

What is claimed is:

1. A system comprising:

a processor configured to receive one or more flowchart diagram files that represent a computer program for a Compound CISC (CCISC) processor;

the processor further configured to identify a flowchart symbol in the one or more flowchart diagram files, to identify a computing category for the flowchart symbol, and to generate one or more CCISC instructions based on the computing category for execution by the CCISC processor, wherein the one or more CCISC instructions are operable to direct the CCISC processor to access and operate on at least two data values in a multi-channel memory during the same clock cycle.

2. The system of claim **1** wherein:

the processor is further configured to identify text within the flowchart symbol that defines one or more computing operations for the computing category, and to generate the one or more CCISC instructions based on the text and the computing category.

3. The system of claim **1** wherein:

the computing category comprises a decision making operation; and

the processor is further configured to generate the one or more CCISC instructions to compare the at least two data values to branch to an address in the program memory.

4. The system of claim **1** wherein:

the computing category comprises a data manipulation operation; and

the processor is further configured to generate the one or more CCISC instructions to change at least one of the data values in the multichannel memory.

5. The system of claim **1** wherein:

the computing category comprises a data movement operation; and

the processor is further configured to generate the one or more CCISC instructions to move at least one data value from a first address to a second address in the multichannel memory.

6. The system of claim **1** wherein:

the processor is further configured to identify a first flowchart symbol in the flowchart diagram file, to identify a second flow chart symbol in the flowchart diagram file, and to identify a process flow based on a direction of an arrow between the first flowchart symbol and the second flow chart symbol;

the processor is further configured to generate a first set of one or more CCISC instructions based on a computing category for the first flowchart symbol, to generate a second set of one or more CCISC instructions based on a computing category for the second flowchart symbol, and to define an execution order for the first and second set of CCISC instructions based on the identified process flow between the first flowchart symbol and the second flowchart symbol.

7. A method comprising:

receiving, by an interface device, one or more flowchart diagram files that represent a computer program for a Compound CISC (CCISC) processor;

identifying, by a flowchart compiler device, a flowchart symbol in the one or more flowchart diagram files;

identifying, by the flowchart compiler device, a computing category for the flowchart symbol; and

generating, by the flowchart compiler device, one or more CCISC instructions based on the computing category for execution by the CCISC processor, wherein the one or more CCISC instructions direct the CCISC processor to access and operate on at least two data values in a multi-channel memory during the same clock cycle.

8. The method of claim **7** wherein:

the method further comprises:

identifying, by the flowchart compiler device, text within the flowchart symbol that defines one or more computing operations for the computing category; and

generating, by the flowchart compiler device, the one or more CCISC instructions further comprises:

generating the one or more CCISC instructions based on the text and the computing category.

9. The method of claim **7** wherein:

the computing category comprises a decision making operation; and generating, by the flowchart compiler device, the one or more CCISC instructions further comprises:

generating the one or more CCISC instructions to compare the at least two data values to branch to an address in the program memory.

10. The method of claim **7** wherein:

the computing category comprises a data manipulation operation; and

generating, by the flowchart compiler device, the one or more CCISC instructions further comprises:

generating the one or more CCISC instructions to change at least one of the data values in the multichannel memory.

11. The method of claim **7** wherein:

the computing category comprises a data movement operation; and

generating, by the flowchart compiler device, the one or more CCISC instructions further comprises:

generating the one or more CCISC instructions to move at least one data value from a first address to a second address in the multichannel memory.

12. The method of claim **7** wherein:

identifying, by the flowchart compiler device, a flowchart symbol further comprises:

identifying a first flowchart symbol in the flowchart diagram file;

identifying a second flow chart symbol in the flowchart diagram file; and

identifying a process flow based on a direction of an arrow between the first flowchart symbol and the second flow chart symbol; and

generating, by the flowchart compiler device, the one or more CCISC instructions further comprises:

generating a first set of one or more CCISC instructions based on a computing category for the first flowchart symbol;

generating a second set of one or more CCISC instructions based on a computing category for the second flowchart symbol; and

defining an execution order for the first and second set of CCISC instructions based on the identified process flow between the first flowchart symbol and the second flowchart symbol.

13. A non-transitory computer readable medium embodying programmed instructions which, when executed by processor, direct the processor to:

receive one or more flowchart diagram files that represent a computer program for a Compound CISC (CCISC) processor;

identify a flowchart symbol in the one or more flowchart diagram files;

identify a computing category for the flowchart symbol; and

generate one or more CCISC instructions based on the computing category for execution by the CCISC processor, wherein the one or more CCISC instructions direct the CCISC processor to access and operate on at least two data values in a multi-channel memory during the same clock cycle.

14. The medium of claim 13 wherein:

the instructions further direct the processor to:

identify text within the flowchart symbol that defines one or more computing operations for the computing category; and

generate the one or more CCISC instructions based on the text and the computing category.

15. The medium of claim 13 wherein:

the computing category comprises a decision making operation; and

the instructions further direct the processor to:

generate the one or more CCISC instructions to compare the at least two data values to branch to an address in the program memory.

16. The medium of claim 13 wherein:

the computing category comprises a data manipulation operation; and

the instructions further direct the processor to:

generate the one or more CCISC instructions to change at least one of the data values in the multichannel memory.

17. The medium of claim 13 wherein:

the computing category comprises a data movement operation; and

the instructions further direct the processor to:

generate the one or more CCISC instructions to move at least one data value from a first address to a second address in the multichannel memory.

18. The medium of claim 13 wherein:

the instructions further direct the processor to:

identify a first flowchart symbol in the flowchart diagram file;

identify a second flow chart symbol in the flowchart diagram file;

identify a process flow based on a direction of an arrow between the first flowchart symbol and the second flow chart symbol;

generate a first set of one or more CCISC instructions based on a computing category for the first flowchart symbol;

generate a second set of one or more CCISC instructions based on a computing category for the second flowchart symbol; and

define an execution order for the first and second set of CCISC instructions based on the identified process flow between the first flowchart symbol and the second flowchart symbol.

* * * * *